¡¡APRUEBA TU EXAMEN CON SCHAUM!!

Programación en C

Schaum

2ª EDICIÓN REVISADA

Byron Gottfried

- REDUCE TU TIEMPO DE ESTUDIO
- 332 EJEMPLOS DETALLADOS, 450 CUESTIONES DE REPASO Y 214 PROBLEMAS DE COMPRENSIÓN
- 705 PROBLEMAS PROPUESTOS
- 8 APÉNDICES QUE RESUMEN LA SINTAXIS DEL LENGUAJE

Utilizado por millones de estudiantes y recomendado por profesores de todo el mundo

Utilízalo para las siguientes asignaturas:

 FUNDAMENTOS DE PROGRAMACIÓN

 PROGRAMACIÓN

 METODOLOGÍA DE LA PROGRAMACIÓN

PROGRAMACIÓN EN C

Segunda edición revisada

PROGRAMACIÓN EN C

Segunda edición revisada

BYRON S. GOTTFRIED
Profesor de Ingeniería Industrial
Universidad de Pittsburgh

Traducción:
JOSÉ RAFAEL GARCÍA LÁZARO
Dpto. Lenguajes y Computación
Universidad de Almería

Revisión técnica:
ALFONSO BOSCH ARÁN
Dpto. Lenguajes y Computación
Universidad de Almería



MADRID • BUENOS AIRES • CARACAS • GUATEMALA • LISBOA • MÉXICO
NUEVA YORK • PANAMÁ • SAN JUAN • SANTAFÉ DE BOGOTÁ • SANTIAGO • SÁO PAULO
AUCKLAND • HAMBURGO • LONDRES • MILÁN • MONTREAL • NUEVA DELHI
PARIS • SAN FRANCISCO • SIDNEY • SINGAPUR • ST. LOUIS • TOKIO • TORONTO

La información contenida en este libro procede de la traducción de la segunda edición en inglés editada por McGraw-Hill Companies, Inc. No obstante, McGraw-Hill no garantiza la exactitud o perfección de la información publicada. Tampoco asume ningún tipo de garantía sobre los contenidos y las opiniones vertidas en dichos textos.

Este trabajo se publica con el reconocimiento expreso de que se está proporcionando una información, pero no tratando de prestar ningún tipo de servicio profesional o técnico. Los procedimientos y la información que se presentan en este libro tienen sólo la intención de servir como guía general.

McGraw-Hill ha solicitado los permisos oportunos para la realización y el desarrollo de esta obra.

PROGRAMACIÓN EN C. Segunda edición revisada



Traducido de la segunda edición en inglés de
PROGRAMMING WITH C

http://www.mcgraw-hill.es
profesional@mcgraw-hill.com

ISBN: 0-07024-035-3

ISBN: 84-481-9846-80-1
Depósito legal: M. 0.000-2005

Editores: Concepción Fernández / Carmelo Sánchez
Compuesto en Puntographic, S. L.

IMPRESO EN MEXICO - PRINTED IN MEXICO

Esta obra se terminó de
Imprimir en Febrero de 2005 en
Programas Educativos, S.A. de C.V.
Calz. Chabacano No. 65- A
Col. Asturias, C.P. 06850, Méx. D.F.
Empresa Certificada por el Instituto Mexicano
de Normalización y Certificación A.C., bajo la
Norma ISO-9002: 1994/NMX-CC-004: 1995 con
el núm. de Registro RSC-048 y bajo la Norma
ISO-14001-1996/NMX-SAA-001: 1998 IMNC
con el núm. de Registro RSAA-003

Contenido

Ejemplos completos de programas

A continuación se listan los ejemplos de programas según el orden en que aparecen en el texto. Los ejemplos van desde los más sencillos a los de complejidad moderada. Se presentan varias versiones de algunos de ellos, en especial de los más sencillos.

1. *Área de un círculo* - Ejemplos 1.6 - 1.13
2. *Conversión de un carácter de minúscula a mayúscula* - Ejemplos 3.31, 7.1
3. *Conversión de texto en minúsculas a mayúsculas* - Ejemplos 4.4, 6.9, 6.12, 6.16, 9.2
4. *Lectura y escritura de una línea de texto* - Ejemplos 4.19, 4.31
5. *Calificaciones medias de los estudiantes* - Ejemplo 4.32
6. *Cálculo del interés compuesto* - Ejemplos 5.1 - 5.4, 8.13
7. *Errores sintácticos* - Ejemplo 5.5
8. *Errores de ejecución (Raíces reales de una ecuación de segundo grado)* - Ejemplo 5.6
9. *Depuración de un programa* - Ejemplo 5.7
10. *Depuración con un depurador interactivo* - Ejemplo 5.8
11. *Generación de cantidades enteras consecutivas* - Ejemplos 6.8, 6.11, 6.14, 6.15
12. *Media de una lista de números* - Ejemplos 6.10, 6.13, 6.17, 6.31
13. *Repetición de la media de una lista de números* - Ejemplo 6.18
14. *Conversión de varias líneas de texto a mayúsculas* - Ejemplos 6.19, 6.34
15. *Codificación de una cadena de caracteres* - Ejemplo 6.20
16. *Cálculos repetidos del interés compuesto con detección de error* - Ejemplo 6.21
17. *Solución de una ecuación algebraica* - Ejemplo 6.22
18. *Cálculo de la depreciación* - Ejemplos 6.26, 7.13
19. *Búsqueda de palíndromos* - Ejemplo 6.32
20. *Mayor de tres cantidades enteras* - Ejemplo 7.9
21. *Cálculo de factoriales* - Ejemplos 7.10, 7.14, 8.2
22. *Simulación de un juego de azar («Shooting Craps»)* - Ejemplos 7.11, 8.9
23. *Escritura inversa* - Ejemplo 7.15
24. *Las torres de Hanoi* - Ejemplo 7.16
25. *Longitud media de varias líneas de texto* - Ejemplos 8.3, 8.5
26. *Búsqueda de un máximo* - Ejemplos 8.4, 8.11
27. *Generación de números de Fibonacci* - Ejemplos 8.7, 8.12, 13.2
28. *Desviaciones respecto a la media* - Ejemplos 9.8, 9.9
29. *Reordenación de una lista de números* - Ejemplos 9.13, 10.16
30. *Un generador de «Pig Latín»* - Ejemplo 9.14
31. *Suma de dos tablas de números* - Ejemplos 9.19, 10.22, 10.24
32. *Reordenación de una lista de cadenas de caracteres* - Ejemplos 9.20, 10.26
33. *Análisis de una línea de texto* - Ejemplo 10.8
34. *Presentación del día del año* - Ejemplo 10.28
35. *Valor futuro de depósitos mensuales (cálculo de intereses compuestos)* - Ejemplos 10.30, 14.13

Prólogo

Desde la publicación de la primera edición de este libro, la popularidad de C ha seguido aumentando. La mayoría de los últimos compiladores aparecidos proporcionan numerosas ampliaciones al estándar ANSI 1989, así como entornos de programación gráficos muy sofisticados que incluyen un depurador, un gestor de proyectos y una ayuda en línea muy detallada. Más aún, el interés de C no ha disminuido por la aparición de C++, debido a que las características de este novedoso lenguaje de programación exigen un sólido conocimiento de C.

Esta segunda edición instruye en el uso del lenguaje C, dentro del contexto del estilo de programación de C contemporáneo. Incluye explicaciones completas y comprensibles de las facetas de C de uso más común, incluidas las del estándar ANSI actual. Además, el libro presenta un enfoque moderno de la programación, insistiendo en la importancia de la claridad, legibilidad, modularidad y eficiencia en el diseño de los programas. Se le presentan al lector de esta forma los principios de la buena programación, así como las reglas específicas del C. Aparecen programas en C completos por todo el libro, a partir del primer capítulo. Se pone especial énfasis en el uso de un estilo de programación interactivo.

El libro puede ser usado por una gran variedad de lectores, desde programadores que estén iniciándose hasta los profesionales expertos. Se adapta bien como libro de texto a un curso de introducción a la programación para estudiantes a nivel universitario, o como un texto complementario, o independientemente, como una eficiente guía.

Se incluyen mucho ejemplos como parte principal del texto. Éstos incluyen a su vez numerosos ejemplos de programación de complejidad variable, así como ilustrativos problemas tipo ejercicio que se adaptan al estándar ANSI C. Muchos de los ejemplos se encuentran resueltos en otros lenguajes de programación en los libros correspondientes de la serie Schaum, proporcionando de esta forma al lector una base para la comparación de varios lenguajes populares.

Al final de cada capítulo aparecen conjuntos de cuestiones de repaso y ejercicios. Las cuestiones de repaso permiten al lector probar la asimilación del material presentado en cada capítulo. Aportan también un eficaz resumen del mismo. Los ejercicios refuerzan los principios presentados en cada capítulo. El lector debería resolver el mayor número posible de estos problemas. Al final del libro aparecen las soluciones a la mayoría de los ejercicios.

Los problemas que requieren la escritura de programas completos escritos en C se presentan al final de cada capítulo, a partir del número cinco. Se insta al lector a escribir y ejecutar el mayor número posible de estos programas. Esto aumentará de forma considerable la confianza en sí mismo y estimulará su interés por el tema. (Para la programación se necesita habilidad, igual que para escribir un libro o tocar un instrumento musical. Esta habilidad no se adquiere simplemente leyendo un libro de texto.)

La mayoría de estos problemas de programación no requieren conocimientos matemáticos o técnicos especiales. Pueden, por tanto, ser resueltos por un amplio espectro de lectores. En caso de utilizar este libro en un curso de programación, es posible que el instructor desee añadir a estos problemas otros que resulten de especial interés en determinadas disciplinas.

Se han realizado una serie de cambios con respecto a la edición anterior. Se ha escrito de nuevo el Capítulo 5, con la utilización del entorno de programación Turbo C++ de Borland

International para ilustrar el uso del C, y se ha redactado de nuevo y ampliado el material correspondiente a las técnicas de depuración. Se han reordenado los tópicos del Capítulo 6 de manera que se corresponden con el orden en que se suelen presentar en la mayoría de los cursos de introducción a la programación, con la bifurcación antes de los bucles. Se ha eliminado del Capítulo 7 parte del material sobre el uso de las funciones, el cual reflejaba un estilo de programación obsoleto, y se ha añadido al Capítulo 10 una sección sobre asignación dinámica de memoria. En la mayoría de los ejemplos de programación se han realizado cambios de estilo; más concretamente, los programas que utilizan varias funciones hacen ahora especial énfasis en el prototipado completo de la función, tal y como recomienda el estándar ANSI actual.

Todos los ejemplos de programación y muchos de los problemas que aparecen al final de los capítulos se han resuelto y ejecutado en un PC («compatible IBM»), utilizando versiones diferentes del compilador Turbo C++ de Borland International.

Las principales características de C están resumidas en los Apéndices del A hasta el H al final del libro. Es recomendable usar este material frecuentemente como referencia rápida. Es particularmente útil al escribir o depurar nuevos programas.

BYRON S. GOTTFRIED

CAPÍTULO 1

Conceptos básicos

Este libro instruye en la programación de computadoras en un lenguaje popular y estructurado: el lenguaje C. Además, veremos cómo analizar problemas que son descritos inicialmente en términos muy generales, cómo esbozarlos y cómo obtener finalmente programas C bien organizados. La multitud de problemas de ejemplo que incluye el texto muestran con detalle estos conceptos.

1.1. INTRODUCCIÓN A LAS COMPUTADORAS

Las computadoras de hoy día se presentan en una amplia variedad de formas. Su rango se extiende desde los *«mainframes»* (grandes computadoras) y *supercomputadoras* masivas y multipropósito hasta las *computadoras personales* de escritorio. Entre ambos extremos nos encontramos con un inmenso conjunto de *minicomputadoras* y *estaciones de trabajo*. Las grandes minicomputadoras se aproximan a los «mainframes» en potencia de cálculo, mientras que las estaciones de trabajo son potentes computadoras personales.

Los «mainframes» y las grandes minicomputadoras se utilizan en muchos negocios, universidades, hospitales y agencias gubernamentales para desarrollar sofisticados cálculos científicos y financieros. Estas computadoras son muy caras (las grandes cuestan millones de dólares) y requieren una plantilla grande de personal para su funcionamiento, así como un entorno especial y cuidadosamente controlado.

Por otro lado, las computadoras personales son pequeñas y baratas. De hecho, en la actualidad muchos estudiantes y profesionales que viajan con frecuencia utilizan mucho computadoras «portátiles» con batería incorporada y con un peso inferior a 2 o 3 kilogramos. En muchas escuelas y empresas se utilizan ampliamente las computadoras personales, que se han convertido en elementos de uso doméstico. La mayoría de los estudiantes utilizan computadoras personales cuando aprenden a programar en C.

La Figura 1.1 muestra a un estudiante trabajando con una computadora portátil.

A pesar de su pequeño tamaño y bajo precio, las modernas computadoras personales rivalizan en potencia de computación con muchas minicomputadoras. Ahora se utilizan para muchas aplicaciones que antes se realizaban en computadoras más grandes y más caras. Más aún, su potencia aumenta a la vez que su precio disminuye progresivamente. El diseño de una computadora personal permite un alto nivel de interacción entre el usuario y la computadora. Muchas aplicaciones (por ejemplo procesadores de texto, programas de tratamiento de gráficos, hojas de cálculo y programas de gestión de bases de datos) están especialmente diseñadas para sacar



Figura 1.1.

partido de este entorno, proporcionando al usuario entrenado una amplia variedad de herramientas creativas para escribir, dibujar o realizar cálculos numéricos. Las aplicaciones con gráficos de alta resolución son también frecuentes.

Muchas veces se conectan computadoras personales con grandes computadoras o con otras computadoras personales, permitiendo su utilización bien como dispositivos autónomos o bien como terminales en una *red* de computadoras. Son frecuentes también las conexiones a través de líneas telefónicas. Dentro de este contexto, podemos ver que las computadoras personales tienden a *complementar* más que a *reemplazar* a las grandes computadoras.

1.2. CARACTERÍSTICAS DE LAS COMPUTADORAS

Todas las computadoras digitales, independientemente de su tamaño, son básicamente dispositivos electrónicos que pueden transmitir, almacenar y manipular *información* (*datos*). Una computadora puede procesar distintos tipos de datos. Esto incluye *datos numéricos*, *alfanuméricos* (nombres, direcciones, etc.), *gráficos* (mapas, dibujos, fotografías, etc.) y *sonido* (música, lectura de textos, etc.). Desde el punto de vista del programador recién iniciado, los dos tipos de datos más familiares son los números y los caracteres. Las aplicaciones científicas y técnicas involucran principalmente datos numéricos, mientras que las aplicaciones empresariales normalmente requieren procesar tanto datos numéricos como alfanuméricos.

Para que la computadora procese un conjunto particular de datos es necesario darle un conjunto apropiado de instrucciones llamado *programa*. Estas instrucciones se introducen en la computadora y se almacenan en una parte de la *memoria* de la máquina.

Un programa almacenado se puede *ejecutar* en cualquier momento. Su ejecución supone los siguientes pasos:

1. Se introduce en la computadora (desde un teclado, un disquete, etc.) un conjunto de información, los *datos de entrada*, y se almacena en una parte de la memoria de ésta.
2. Los datos de entrada se procesarán para producir ciertos resultados deseados, los *datos de salida.*
3. Los datos de salida, y probablemente algunos de los datos de entrada, se imprimirán en papel o se presentarán en un *monitor* (una pantalla diseñada especialmente para visualizar salida de computadora).

Este procedimiento de tres pasos se puede repetir tantas veces como se desee, procesando rápidamente una gran cantidad de datos. En cualquier caso, se debe tener presente que estos pasos, especialmente el 2 y el 3, pueden ser largos y complicados.

EJEMPLO 1.1. Una computadora ha sido programada para calcular el área de un círculo utilizando la fórmula $a = \pi r^2$, dando el valor numérico del radio como dato de entrada. Es necesario seguir estos pasos:

1. Leer el valor numérico del radio del círculo.
2. Calcular el valor del área utilizando la fórmula anterior. Este valor se almacenará, junto con los datos de entrada, en la memoria de la computadora.
3. Imprimir (presentar en el terminal) los valores del radio y del área correspondiente.
4. Parar.

Cada uno de estos pasos requiere una instrucción o más en un programa.

Lo anteriormente dicho ilustra dos características importantes de una computadora digital: *memoria* y *capacidad de ser programada*. Otras características importantes son su *velocidad y fiabilidad*. Hablaremos más de memoria, velocidad y fiabilidad en los siguientes párrafos. La programabilidad será discutida ampliamente en el resto del libro.

Memoria

Cada fragmento de información almacenado en la memoria de la computadora es codificado como una combinación única de ceros y unos. Estos ceros y unos se llaman *bits* (*dígitos binarios*). Un dispositivo electrónico representa cada bit, en cierto sentido, como «apagado» (cero) o «encendido» (uno).

Las computadoras personales tienen la memoria organizada en grupos de 8 bits, denominados *bytes*, como se muestra en la Figura 1.2. Hay que advertir que cada bit está numerado, empezando por 0 (el bit del extremo derecho) y terminando en 7 (el bit del extremo izquierdo). Normalmente, un carácter (por ejemplo una letra, un solo dígito o un símbolo de puntuación) ocupará un byte de memoria. Una instrucción puede ocupar 1, 2 o 3 bytes. Una cierta cantidad numérica puede ocupar de 1 a 8 bytes, dependiendo de la *precisión* (el número de cifras significativas) y el *tipo* (entero, coma flotante, etc.).

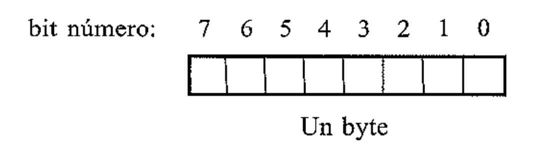


Figura 1.2.

Normalmente se expresa el tamaño de la memoria computadora como algún múltiplo de 2^{10} = 1024 by-tes; esto es, 1K. Las computadoras personales actuales tienen memorias de tamaños comprendidos típicamente entre 256 y 1024 megabytes, siendo 1 megabyte (1M) equivalente a $2^{10} \cdot 2^{10}$ bytes, o 2^{10}K = 1024 Kbytes.

EJEMPLO 1.2. La memoria de una computadora personal tiene una capacidad de 16Mbytes. Así pues, se pueden almacenar en su memoria 16 · 1024 · 1024 = 16 777 216 caracteres y/o instrucciones. Si se utiliza la memoria completa para representar caracteres (lo cual es poco probable), entonces se podrán almacenar unos 200 000 nombres y direcciones a la vez, suponiendo unos 80 caracteres por cada nombre y dirección.

Si se utiliza la memoria para almacenar datos numéricos en vez de nombres y direcciones, se podrán almacenar más de 4 millones de números a la vez, suponiendo que cada cantidad numérica requiera 4 bytes de memoria.

Las grandes computadoras tienen memorias organizadas en *palabras* en lugar de bytes. Cada palabra tendrá un número relativamente grande de bits, típicamente 32 o 36. En la Figura 1.3 aparece la organización a nivel de bits de una palabra de 32 bits. Nótese que los bits están numerados, comenzando por el 0 (el bit del extremo derecho) y finalizando en 31 (el bit del extremo izquierdo).

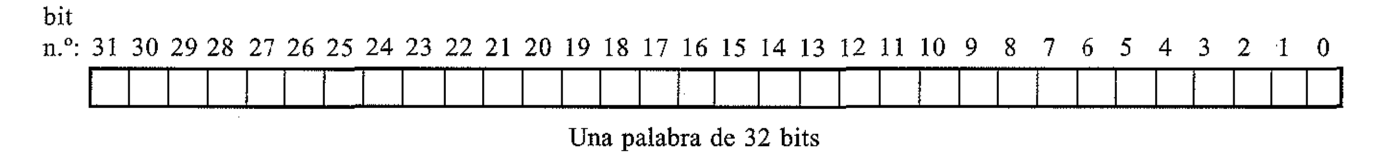


Figura 1.3.

La Figura 1.4 muestra la misma palabra de 32 bits organizada en cuatro bytes consecutivos. Los bytes están numerados de la misma forma que los bits individuales, de 0 (el byte del extremo derecho) a 3 (el byte del extremo izquierdo).

El uso de palabras de 32 o 36 bits permite representar una cantidad numérica o un pequeño *grupo* de caracteres en una sola palabra de memoria. Normalmente las grandes computadoras tienen varios millones de palabras (varias megapalabras) de memoria.

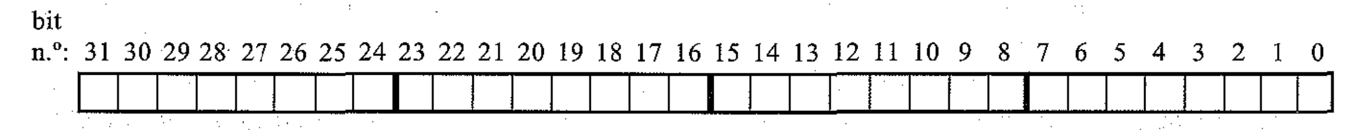


Figura 1.4.

EJEMPLO 1.3. La memoria de una gran computadora tiene una capacidad de 32M (32 768K) palabras, lo que es equivalente a 32 · 1024 · 1024 = 33 554 432 palabras. Si se usa la memoria entera para representar datos numéricos (lo cual no es muy probable), entonces se podrán almacenar en la computadora más de 33 millones de números a la vez, suponiendo que cada cantidad numérica requiera una palabra de memoria.

Si se utiliza la memoria completa para representar caracteres en lugar de datos numéricos, se podrán almacenar más de 130 millones de caracteres a la vez, suponiendo 4 caracteres por palabra. Esta memoria es suficiente para almacenar los contenidos de varios libros voluminosos.

La mayoría de las computadoras utilizan también *dispositivos auxiliares de almacenamiento* (por ejemplo cintas magnéticas, discos, dispositivos ópticos de memoria) además de la memoria principal. Estos dispositivos pueden almacenar gigabytes (1G = 1024M) de información. Además, permiten que se grabe permanentemente la información, pudiendo normalmente ser desconectados de la computadora y almacenados cuando no se utilicen. Sin embargo, el tiempo de acceso (el tiempo necesario para almacenar o recuperar información) de estos dispositivos auxiliares suele ser considerablemente mayor que el de la memoria principal de la computadora.

Velocidad y fiabilidad

Gracias a su velocidad extremadamente alta, una computadora puede efectuar en unos pocos minutos cálculos que requerirían muchos días, posiblemente meses o años, si se hiciesen a mano. Por ejemplo, se pueden procesar las calificaciones de final de semestre de todos los estudiantes de una gran universidad en unos pocos minutos de tiempo de cómputo.

El tiempo requerido para efectuar tareas de cálculo sencillas, como sumar dos números, se expresa habitualmente en *microsegundos* ($1\ \mu seg = 10^{-6}$ seg) o *nanosegundos* ($1\ nseg = 10^{-3}\ \mu seg = 10^{-9}$ seg). Por tanto, si una computadora puede sumar dos números en 10 nanosegundos (típico de una computadora media actual), puede efectuar 100 millones (10^8) de sumas en un segundo.

Esta alta velocidad es acompañada por un nivel equivalente de fiabilidad. Una computadora nunca cometerá un error por sí misma. Los tan divulgados «errores de computadoras», tales como que a una persona se le cargue un recibo de teléfono de varios millones de dólares, son el resultado de errores de programación o de errores en la introducción de datos, y no causados por la propia computadora.

1.3. MODOS DE OPERACIÓN

Una gran computadora puede ser compartida por muchos usuarios de dos formas diferentes. Éstas son el *modo de procesamiento por lotes* y el *modo interactivo*. Cada modo tiene sus propias ventajas para ciertos tipos de problemas.

Procesamiento por lotes

En el *procesamiento por lotes* se cargan una serie de tareas en la computadora, se almacenan internamente y luego se procesan secuencialmente. (Una *tarea* es un programa y sus conjuntos de datos de entrada asociados.) Tras ser procesada la tarea, una impresora de alta velocidad imprime en múltiples hojas de papel la salida junto con el listado del programa. Normalmente el

usuario recogerá la salida impresa tras un cierto tiempo transcurrido desde que la tarea haya sido procesada.

En el *procesamiento por lotes clásico* (hoy día obsoleto) el programa y los datos eran grabados en *tarjetas perforadas*. Esta información se introducía en la computadora mediante un lector mecánico de tarjetas y a continuación se procesaba. En los albores de la informática, todas las tareas se procesaban de esta manera.

El *procesamiento por lotes moderno* va unido generalmente a sistemas de tiempo compartido (ver próxima sección). En estos sistemas el programa y los datos son introducidos en la computadora mediante un *terminal* o una computadora personal. Se almacena entonces la información en la memoria de la computadora y es procesada en un determinado orden. Esta forma de procesamiento por lotes es preferible a la clásica, ya que elimina la necesidad de utilizar tarjetas perforadas y permite la edición de la información de entrada (programa y datos) mientras se introduce.

En el procesamiento por lotes pueden circular muy rápidamente grandes cantidades de información (programas y datos) hacia dentro y fuera de la computadora. Además, el usuario no necesita estar presente mientras se procesa la tarea. Este modo de operación se adecúa a tareas que requieren gran cantidad de tiempo de cómputo o que son, en general, muy largas. Por otra parte, el tiempo total requerido para que una tarea sea procesada de esta forma puede variar entre varios minutos y varias horas, aun cuando la tarea pueda sólo necesitar uno o dos segundos de tiempo real de cómputo. (Cada tarea debe esperar su turno hasta que es cargada, procesada y escrita su salida.) De esta forma, el procesamiento por lotes puede ser poco deseable cuando es necesario procesar muchas pequeñas tareas y presentar sus resultados lo antes posible (como, por ejemplo, cuando se aprende a programar).

Sistemas de tiempo compartido

Un sistema de *tiempo compartido* permite a diferentes usuarios utilizar una sola computadora a la vez. La computadora anfitriona puede ser un «mainframe», una minicomputadora o una gran computadora de sobremesa. Los usuarios se comunican con la computadora a través de sus terminales individuales. En las redes de tiempo compartido modernas es frecuente utilizar las computadoras personales como terminales. La computadora anfitriona puede soportar muchos terminales a la vez, ya que trabaja mucho más rápido que un operador humano en su terminal. De este modo, cada usuario será ajeno a la presencia de otros usuarios y creerá tener toda la computadora anfitriona a su disposición.

Cada terminal individual puede estar bien conectado directamente a la computadora central, o bien a través de las líneas telefónicas, de un circuito de microondas o incluso un satélite espacial. El terminal puede, por tanto, estar localizado lejos —quizás cientos de kilómetros— de su computadora central. Son particularmente frecuentes los sistemas en que las computadoras personales se conectan a grandes «mainframes» a través de líneas telefónicas. Tales sistemas hacen uso de *módems* (dispositivos *mod*uladores/*dem*oduladores) para convertir las señales digitales de la computadora en señales telefónicas analógicas y viceversa. A través de dicha configuración, una persona que trabaja en casa puede fácilmente acceder con su computadora personal a la computadora remota del colegio o de la oficina.

Un sistema de tiempo compartido es más adecuado para el procesamiento de tareas relativamente sencillas que no requieran de la transmisión de muchos datos o de grandes cantidades de

tiempo de cómputo. Muchas aplicaciones de las escuelas y de las oficinas comerciales presentan estas características. Utilizando el tiempo compartido se pueden procesar tales aplicaciones de una forma rápida, fácil y barata.

EJEMPLO 1.4. Una gran universidad dispone de una computadora de tiempo compartido con capacidad para 200 terminales y 80 conexiones telefónicas separadas. Los terminales situados en diferentes lugares del campus están conectados directamente a un gran «mainframe». Cada terminal es capaz de transmitir información a o desde la computadora central a una velocidad máxima de 960 caracteres por segundo.

Las conexiones telefónicas permiten a estudiantes que no se encuentran en el campus conectar sus computadoras personales a la computadora central. Cada computadora personal puede transmitir datos a o desde la computadora central a una velocidad máxima de 240 caracteres por segundo. Así pues, los 280 terminales y computadoras pueden interactuar con la computadora central a la vez, estando cada estudiante despreocupado de que los demás están compartiendo al mismo tiempo la computadora.

Computación interactiva

La *computación interactiva* es un tipo de entorno de computación que surgió con los sistemas de tiempo compartido comerciales y que ha sido mejorado con el amplio uso de las computadoras personales. En un entorno de computación interactivo, durante la sesión de trabajo existe interacción entre el usuario y la computadora. De este modo, el usuario será consultado periódicamente para que proporcione cierta información, la cual determinará las acciones pertinentes a realizar por la computadora y viceversa.

EJEMPLO 1.5. Un estudiante desea usar una computadora personal para calcular el radio de un círculo cuya área tiene el valor de 100. Se dispone de un programa que calcula el área del círculo, dado el radio. (Nótese que esto es justo lo contrario de lo que el estudiante desea hacer.) Este programa no es exactamente lo que necesita, pero permite al estudiante proceder por el método de prueba y error. El procedimiento consistirá en adivinar un valor para el radio y luego calcular el área correspondiente. El procedimiento de prueba y error continuará hasta que el estudiante encuentre un valor del radio para que el área sea lo suficientemente próxima a 100.

Una vez que comienza la ejecución del programa, aparece el mensaje

```
Radio = ?
```

Entonces el estudiante introduce un valor del radio. Supongamos que da un valor de 5. La computadora responderá presentando:

```
Area = 78.5398
¿Deseas repetir el cálculo?
```

El estudiante responde `sí` o `no`. Si el estudiante dice `sí`, entonces aparece de nuevo el mensaje:

```
Radio = ?
```

y se repetirá de nuevo el proceso. Si el estudiante dice `no`, entonces se presenta en la pantalla:

```
Adiós
```

y finaliza la ejecución del programa.

A continuación se muestra una copia impresa de la información presentada durante una sesión interactiva típica, usando el programa antes descrito. En esta sesión se determina un valor aproximado de `r = 5.6` después de sólo tres cálculos. Se ha subrayado la información aportada por el estudiante.

```
Radio = ? 5
Area = 78.5398

¿Deseas repetir el cálculo? sí

Radio = ? 6
Area = 113.097

¿Deseas repetir el cálculo? sí

Radio = ? 5.6
Area = 98.5204

¿Deseas repetir el cálculo? no

Adiós
```

Nótese la forma en que parecen conversar la computadora y el estudiante. Observe también cómo el estudiante espera hasta que ve el valor del área calculado antes de decidir si continúa o no haciendo cálculos. Si se inicia otro nuevo cálculo, el nuevo valor del radio que proporcione el estudiante dependerá de los resultados de cálculos anteriores.

A veces los programas diseñados para aplicaciones de tipo interactivo se denominan *conversacionales*. Los juegos de computadoras son excelentes ejemplos de aplicaciones de este tipo. En ellos aparecen elaborados gráficos y acciones rápidas, aun cuando las respuestas del usuario son más de tipo reflejo que numéricas o verbales.

1.4. TIPOS DE LENGUAJES DE PROGRAMACIÓN

Se pueden utilizar muchos lenguajes para programar una computadora. El más básico es el *lenguaje máquina*, una colección de instrucciones muy detalladas y crípticas que controlan la circuitería interna de la máquina. Éste es el dialecto natural de la computadora. Muy pocos programas se escriben actualmente en lenguaje máquina por dos razones importantes: primero, porque el lenguaje máquina es muy incómodo para trabajar, y segundo, porque la mayoría de las máquinas tienen sus repertorios de instrucciones propios. Así, un programa escrito en lenguaje máquina para una computadora no puede ser ejecutado en otra de distinto tipo sin modificaciones importantes.

Lo más frecuente es utilizar lenguajes de *alto nivel*, cuyas instrucciones son más compatibles con los lenguajes y la forma de pensar humanos. La mayoría son lenguajes de *propósito general*, como C. (Otros lenguajes de propósito general son Pascal, Fortran y BASIC.) Hay también lenguajes de *propósito especial* que están diseñados específicamente para algún tipo particular de aplicación. Algunos ejemplos comunes son CSMP y SIMAN, que son lenguajes orientados a la *simulación*, y LISP, un lenguaje orientado al *tratamiento de listas* que se utiliza ampliamente en aplicaciones de inteligencia artificial.

Por norma general, una sola instrucción de un lenguaje de alto nivel será equivalente a varias de lenguaje máquina. Esto simplifica enormemente la tarea de escribir programas completos y

correctos. Además, las reglas de programación en un lenguaje de alto nivel se pueden aplicar a todas las computadoras, de manera que un programa escrito para una computadora se puede ejecutar normalmente en otras máquinas diferentes con muy pocas modificaciones o directamente. Por tanto, el uso de un lenguaje de alto nivel ofrece tres ventajas importantes respecto al lenguaje máquina: *sencillez*, *uniformidad* y *portabilidad* (independencia de la máquina).

En todo caso, un programa escrito en lenguaje de alto nivel ha de ser traducido a lenguaje máquina antes de poder ser ejecutado. Esto se conoce como *compilación* o *interpretación*, dependiendo de cómo se lleve a cabo. (Los compiladores traducen el programa completo a lenguaje máquina antes de ejecutar cualquiera de las instrucciones. Los intérpretes, por otro lado, recorren el programa tomando instrucciones una a una en pequeños grupos que traducen y ejecutan.) En cualquier caso, la traducción se lleva a cabo de forma automática por la computadora. De hecho, los programadores recién iniciados a veces no se dan cuenta de que este hecho está ocurriendo, ya que típicamente sólo ven el programa original en alto nivel, los datos de entrada y los resultados obtenidos. La mayoría de las implementaciones de C operan como compiladores.

Un compilador o intérprete es a su vez un programa. Acepta como datos de entrada un programa en alto nivel (por ejemplo un programa en C) y genera como resultado el correspondiente programa en lenguaje máquina. El programa original en lenguaje de alto nivel se llama programa *fuente*, y el programa resultante en lenguaje máquina se llama programa *objeto*. Cada computadora debe disponer de su propio compilador o intérprete para cada lenguaje de alto nivel particular.

Es más conveniente, por norma general, desarrollar un programa nuevo utilizando un intérprete en vez de un compilador. Una vez que se ha conseguido el programa sin errores, una versión compilada se ejecutará normalmente de forma mucho más rápida que una interpretada. Las razones por las que ocurre esto quedan fuera del ámbito de la presente discusión.

1.5. INTRODUCCIÓN AL C

C es un lenguaje de programación estructurado de propósito general. Sus instrucciones constan de términos que se parecen a expresiones algebraicas, además de ciertas *palabras clave* inglesas como `if`, `else`, `for`, `do` y `while`. En este sentido, C recuerda a otros lenguajes de programación estructurados como Pascal y Fortran. C tiene también algunas características adicionales que permiten su uso a un nivel más bajo, cubriendo así el vacío entre el lenguaje máquina y los lenguajes de alto nivel más convencionales. Esta flexibilidad permite el uso de C en la *programación de sistemas* (por ejemplo, para el diseño sistemas operativos) así como en la *programación de aplicaciones* (por ejemplo, para redactar un programa que resuelva un complicado sistema de ecuaciones matemáticas o un programa que escriba las facturas para los clientes).

C se caracteriza por hacer posible la redacción de programas fuente muy concisos, debido en parte al gran número de operadores que incluye el lenguaje. Tiene un repertorio de instrucciones relativamente pequeño, aunque las implementaciones actuales incluyen numerosas *funciones de biblioteca* que mejoran las instrucciones básicas. Es más, el lenguaje permite a los usuarios escribir funciones de biblioteca adicionales para su propio uso. De esta forma, las características y capacidades del lenguaje pueden ser ampliadas fácilmente por el usuario.

Hay compiladores de C disponibles para computadoras de todos los tamaños, y los intérpretes de C se están haciendo cada vez más comunes. Los compiladores son frecuentemente compactos y generan programas objeto que son pequeños y muy eficientes en comparación con los

programas generados a partir de otros lenguajes de alto nivel. Los intérpretes son menos eficientes, aunque resultan de uso más cómodo en el desarrollo de nuevos programas. Muchos programadores comienzan utilizando un intérprete, y una vez que se ha depurado el programa (eliminado los errores del mismo) utilizan un compilador.

Otra característica importante de C es que los programas son muy portables, más que los escritos en otros lenguajes de alto nivel. La razón de esto es que C deja en manos de las funciones de biblioteca la mayoría de las características dependientes de la computadora. Toda versión de C se acompaña de su propio conjunto de funciones de biblioteca, que están escritas para las características particulares de la computadora en la que se instale. Estas funciones de biblioteca están relativamente normalizadas y el acceso a cada función de biblioteca es idéntico en todas las versiones de C. De esta forma, la mayoría de los programas en C se pueden compilar y ejecutar en muchas computadoras diferentes prácticamente sin modificaciones.

Historia del C

C fue desarrollado originalmente en los años setenta por Dennis Ritchie en Bell Telephone Laboratories, Inc. (ahora una sucursal de AT&T). Es el resultado de dos lenguajes anteriores, el BCPL y el B, que se desarrollaron también en los laboratorios Bell. C estuvo confinado al uso en los laboratorios Bell hasta 1978, cuando Brian Kernighan y Ritchie publicaron una descripción definitiva del lenguaje *. La definición de Kernighan y Ritchie se denomina frecuentemente «K&R C».

Tras la publicación de la definición de K&R, los profesionales de las computadoras, impresionados por las muchas características deseables del C, comenzaron a promover el uso del lenguaje. A mediados de los ochenta la popularidad del C se había extendido por todas partes. Se habían escrito numerosos compiladores e intérpretes de C para computadoras de todos los tamaños y se habían desarrollado numerosas aplicaciones comerciales. Es más, muchas aplicaciones que se habían escrito originalmente en otros lenguajes se reescribieron en C para tomar partido de su eficiencia y portabilidad.

Las primeras implementaciones comerciales de C diferían en parte de la definición original de Kernighan y Ritchie, creando pequeñas incompatibilidades entre las diferentes implementaciones del lenguaje. Estas diferencias reducían la portabilidad que el lenguaje intentaba proporcionar. Consecuentemente, el Instituto Nacional Americano de Estándares ** (comité ANSI X3J11) desarrolló una definición estandarizada del lenguaje C. La mayoría de los compiladores e intérpretes comerciales de C actuales adoptan el estándar ANSI. Algunos compiladores también pueden proporcionar características adicionales propias.

En la década de los ochenta, Bjarne Stroustrup *** desarrolló en los laboratorios Bell otro lenguaje de programación de alto nivel denominado C++. Éste se basa en C, y por tanto todas las características estándar de C están disponibles en C++. Sin embargo, C++ no es una mera extensión de C. Incorpora nuevos fundamentos que constituyen una base para la *programación orientada a objetos* —un nuevo paradigma de la programación de interés para los programadores

* Brian W. Kernighan y Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978.

** Estándar ANSI X3.159-1989. Instituto Nacional Americano de Estándares, 1430 Broadway, New York, NY, 10018. (Ver también Brian W. Kernighan y Dennis M. Ritchie, *The C Programming Language*, 2.ª edición, Prentice-Hall, 1988.)

*** Stroustrup, Bjarne, *The C Programming Language*, segunda edición, Addison-Wesley, 1991.

profesionales—. En este libro no se describe C++, pero sí hay que indicar que un sólido conocimiento de C es un buen punto de partida para aprender C++.

Este libro describe las características de C incluidas en el estándar ANSI y soportadas por los compiladores e intérpretes comerciales de C. El lector que asimile todo este material no tendrá dificultad en adaptar un programa en C a una versión particular del lenguaje.

Estructura de un programa en C

Todo programa en C consta de uno o más módulos llamados *funciones*. Una de las funciones se llama `main`. El programa siempre comenzará por la ejecución de la función `main`, la cual puede acceder a las demás funciones. Las definiciones de las funciones adicionales se deben realizar aparte, bien precediendo o siguiendo a `main` (encontrará más información sobre esto en los Capítulos 7 y 8).

Cada función debe contener:

1. Una *cabecera* de la función, que consta del nombre de la función, seguido de una lista opcional de *argumentos* encerrados entre paréntesis.
2. Una lista de *declaración* de argumentos, si se incluyen éstos en la cabecera.
3. Una *instrucción compuesta*, que contiene el resto de la función.

Los argumentos son símbolos que representan información que se le pasa a la función desde otra parte del programa. (También se llama *parámetros* a los argumentos.)

Cada instrucción compuesta se encierra con un par de llaves, { }. Las llaves pueden contener combinaciones de instrucciones elementales (denominadas *instrucciones de expresión*) y otras instrucciones compuestas. Así las instrucciones compuestas pueden estar anidadas, una dentro de otra. Cada instrucción de expresión debe acabar en punto y coma (;).

Los *comentarios* pueden aparecer en cualquier parte del programa, mientras estén situados entre los delimitadores `/*` y `*/` (por ejemplo: `/* esto es un comentario */`). Los comentarios son útiles para identificar los elementos principales de un programa o para explicar la lógica subyacente de éstos.

Estos componentes del programa se discutirán más tarde en este libro con mucho más detalle. Por ahora sólo se expone una visión general de las características básicas que caracterizan la mayoría de los programas en C.

EJEMPLO 1.6. Área de un círculo. He aquí un programa en C elemental que lee el radio de un círculo, calcula su área y escribe le resultado calculado.

```
/*programa para calcular
  el área de un círculo*/      /* TITULO (COMENTARIO) */

#include <stdio.h>             /* ACCESO A ARCHIVO DE BIBLIOTECA */

main()                         /* CABECERA DE FUNCION */
{
  float radio, area;           /* DECLARACION DE VARIABLES */

  printf("Radio = ? ");        /* INSTRUCCION DE SALIDA */
  scanf("%f", &radio);         /* INSTRUCCION DE ENTRADA */
```

```
float procesar(float r)       /* definición de función */
{
   float a;                    /* declaración de variable local */

   a = PI * r * r;
   return(a);
}
```

Esta versión utiliza una función definida por el programador, la función `procesar`, que se ocupa de efectuar realmente los cálculos. Dentro de esta función, `r` es un argumento (o *parámetro*) que representa el valor del radio proporcionado desde `main`, y `a` es el resultado calculado que se devuelve a `main`. Aparece una referencia a la función en `main`, dentro de la instrucción

```
area = procesar(radio);
```

Una *declaración de función* precede a la función `main`, que indica que `procesar` acepta un argumento en coma flotante y devuelve también un valor en coma flotante. El uso de funciones se discutirá con detalle en el Capítulo 7.

Este programa también contiene una *constante simbólica*, `PI`, que representa el valor numérico `3.14159`. Ésta es una forma de representación que existe para conveniencia del programador. Cuando se compila el programa, la constante simbólica se reemplaza automáticamente por su valor numérico.

Cuando se ejecuta este programa, se comporta de la misma forma que el programa del Ejemplo 1.6.

EJEMPLO 1.8. Área de un círculo con comprobación de error. Consideremos ahora una variación del programa dado en el Ejemplo 1.7.

```
/* programa para calcular el área de un círculo
   con comprobación de error */

#include <stdio.h>

#define PI 3.14159

float procesar(float radio);   /* prototipo de función */

main()
{
   float radio, area;          /* declaración de variables */

   printf("Radio = ? ");
   scanf("%f", &radio);

   if (radio < 0)
      area = 0;
   else
      area = procesar(radio);

   printf("Area = %f", area);
}
```

```
float procesar(float r)    /* definición de función */

{

   float a;                /* declaración de variable local */

   a = PI * r * r;
   return(a);

}
```

Este programa calcula de nuevo el área de un círculo. Incluye la función `procesar` y la constante simbólica `PI`, como se discutió en el ejemplo anterior. Hemos añadido una sencilla rutina de corrección de errores, que comprueba si el valor del radio es menor que cero. (Matemáticamente, un valor negativo del radio no tiene sentido.) La comprobación se lleva a cabo dentro de `main`, utilizando una instrucción `if-else` (ver sección 6.6). De esta forma, si `radio` tiene un valor negativo, se le asigna a `area` el valor cero; en cualquier otro caso, se calcula el valor de `area` en `procesar`, como vimos anteriormente.

EJEMPLO 1.9. Áreas de varios círculos. Ampliando los anteriores programas hemos redactado éste, en el que se calculan las áreas de varios círculos.

```
/* programa para calcular áreas de varios círculos
   usando un bucle for */

#include <stdio.h>

#define PI 3.14159

float procesar(float radio);  /* prototipo de función */

main()

{
   float radio, area;         /* declaración de variables */
   int cont, n;               /* declaración de variables */

   printf("Nº de círculos? ");
   scanf("%d", &n);

   for (cont = 1; cont <= n; ++cont) {
      printf("\nCírculo nº %d:  Radio = ? ", cont);
      scanf("%f", &radio);

      if (radio < 0)
         area = 0;
      else
         area = procesar(radio);

      printf("Area = %f\n", area);

   }
}
```

```
float procesar(float r)      /* definición de función */

{
   float a;                    /* declaración de variable local */

   a = PI * r * r;
   return(a);

}
```

En este caso el número total de círculos, representado por la variable entera `n`, se debe introducir antes de que se realice cualquier cálculo. Se utiliza entonces la instrucción `for` para calcular iterativamente las áreas de los `n` círculos (ver sección 6.4).

Observe el uso de la variable `cont`, que se utiliza como un contador dentro del bucle `for` (dentro de la porción del programa que se repite). El valor de `cont` se incrementará en 1 en cada pasada por el bucle. Nótese también la expresión `++cont` que aparece en la instrucción `for`. Ésta es una notación abreviada para incrementar el valor del contador en 1; de hecho es equivalente a `cont = cont + 1` (ver sección 3.2).

Cuando se ejecuta el programa, genera un diálogo interactivo como el que se muestra a continuación. Las respuestas del usuario están, de nuevo, subrayadas.

```
Nº de círculos? 3

Círculo nº 1:   Radio = ? 3
Area = 28.274309

Círculo nº 2:   Radio = ? 4
Area = 50.265442

Círculo nº 3:   Radio = ? 5
Area = 78.539749
```

EJEMPLO 1.10. Áreas de un número indeterminado de círculos. El programa anterior se puede mejorar de manera que procese un número indeterminado de círculos, haciendo que continúen los cálculos hasta que se introduzca un valor cero para el radio. Esto evita la necesidad de contar, además de especificar por adelantado el número de círculos. Esto es especialmente útil cuando hay que procesar una gran cantidad de datos.

Éste es el programa completo.

```
/* programa para calcular áreas de varios círculos, usando
   un bucle for; no se especifica el número de círculos */

#include <stdio.h>

#define PI 3.14159
```

```
float procesar(float radio); /* prototipo de función */

main()

{

  float radio, area;            /* declaración de variables */
  int cont;                     /* declaración de variables */

  printf("Para PARAR, introducir 0 en el valor del radio\n");
  printf("\nRadio = ? ");
  scanf("%f", &radio);

  for (cont = 1; radio != 0; ++cont) {

      if (radio < 0)
         area = 0;
      else
         area = procesar(radio);

      printf("Area = %f\n", area);

      printf("\nRadio = ? ");
      scanf("%f", &radio);
  }

}

float procesar(float r)       /* definición de función */

{
  float a;                      /* declaración de variable local */

  a = PI * r * r;
  return(a);
}
```

Nótese que este programa presentará un mensaje al comienzo de la ejecución, informando al usuario de cómo finalizar los cálculos.

Abajo se muestra el diálogo resultante de una ejecución típica de este programa. Una vez más, las respuestas del usuario están subrayadas.

```
Para PARAR, introducir 0 en el valor del radio

Radio = ? 3
Area = 28.274309

Radio = ? 4
Area = 50.265442

Radio = ? 5
Area = 78.539749

Radio = ? 0
```

EJEMPLO 1.11. Áreas de un número indeterminado de círculos. He aquí una variante del programa mostrado en el ejemplo anterior.

```
/* programa para calcular áreas de varios círculos, usando
   un bucle while; no se especifica el número de círculos */

#include <stdio.h>

#define PI 3.14159

float procesar(float radio);     /* prototipo de función */

main()

{
   float radio, area;             /* declaración de variables */

   printf("Para PARAR, introducir 0 en el valor del radio\n");
   printf("\nRadio = ? ");
   scanf("%f", &radio);

   while (radio != 0) {

        if (radio < 0)
            area = 0;
        else
            area = procesar(radio);

        printf("Area = %f\n", area);

        printf("\nRadio = ? ");
        scanf("%f", &radio);
    }

}

float procesar(float r)       /* definición de función */

{

   float a;                   /* declaración de variable local */

   a = PI * r * r;
   return(a);

}
```

Este programa realiza la misma función que el mostrado en el ejemplo anterior. Sin embargo, hemos usado ahora una instrucción `while` en vez de una `for` para llevar a cabo la ejecución repetida del programa (ver sección 6.2). La instrucción `while` seguirá ejecutándose mientras se le asigne a `radio` un valor que no sea cero.

En general, la instrucción while continuará ejecutándose mientras la expresión contenida en los paréntesis se considere *verdadera*. De esta forma se puede escribir más brevemente la instrucción `while` como:

```
while (radio) {
```

en lugar de

```
while (radio != 0) {
```

porque cualquier valor de `radio` no nulo se interpretará como una condición *verdadera*.

Algunos problemas se adecúan mejor a la utilización de la instrucción `for`, mientras que otros al uso de `while`. La instrucción `while` es algo más sencilla en esta aplicación en particular. Hay una tercera instrucción para generar bucles, la instrucción `do-while`, que es similar a la `while` mostrada anteriormente. (Más sobre esto en el Capítulo 6).

Cuando se ejecuta este programa, genera un diálogo interactivo idéntico al que se mostró en el Ejemplo 1.10.

EJEMPLO 1.12. Cálculo y almacenamiento del área de varios círculos. Algunos problemas requieren que se almacene en la computadora una serie de resultados ya calculados, quizá para ser utilizados en cálculos posteriores. Los datos de entrada correspondientes se pueden también almacenar internamente junto con los resultados calculados. Esto se puede llevar a cabo mediante el uso de *arrays*.

El siguiente programa utiliza dos arrays, llamados `radio` y `area`, para almacenar el radio y el área de hasta `100` círculos. Cada array se puede ver como una lista de números. Cada uno de los números individuales dentro de cada lista es un *elemento de array*. Los elementos de un array están numerados, empezando por 0. Así el radio del primer círculo se almacenará dentro del elemento `radio[0]`, el radio del segundo círculo se almacenará en `radio[1]`, y así sucesivamente. De igual forma, las áreas correspondientes se almacenarán en `area[0]`, `area[1]`, etc.

Éste es el programa completo.

```
/* programa para calcular áreas de varios círculos, usando
   un bucle while; los resultados se almacenan en un array;
   no se especifica el número de círculos */

#include <stdio.h>

#define PI 3.14159

float procesar(float radio);  /* prototipo de función */

main()

{

   int n, i = 0;              /* declaración de variables */
   float radio[100],
         area[100];           /* declaración de arrays */

   printf("Para PARAR, introducir 0 en el valor del radio\n\n");
   printf("Radio = ? ");
   scanf("%f", &radio[i]);
   while (radio[i]) {

       if (radio[i] < 0)
           area[i] = 0;
```

```
        else
            area[i] = procesar(radio[i]);

        printf("Radio = ? ");
        scanf("%f", &radio[++i]);

    }

    n = --i;        /* El mayor valor de i */

    /* presentar los elementos del array */
    printf("\nRelación de resultados\n\n");
    for (i = 0; i <= n; ++i)
        printf("Radio = %f    Area = %f\n",radio[i], area[i]);
}

float procesar(float r)        /* definición de función */

{
    float a;                   /* declaración de variable local */

    a = PI * r * r;
    return(a);
}
```

Como en ejemplos anteriores, se introducirá un número indeterminado de radios. Se introduce cada valor del radio y se almacena el valor i-ésimo de `radio[i]`. Se calcula entonces el área correspondiente y se almacena en `area[i]`. Este proceso continuará hasta que se introduzcan todos los radios y al final se proporcione el valor 0 para el radio. Se presenta finalmente el conjunto entero de valores almacenados (elementos del array cuyos valores no son cero).

Observemos la aparición de la expresión `++i` dos veces en el programa. Cada una de estas expresiones hace que se incremente en 1 el valor de `i`; de hecho son equivalentes a `i=i+1`. Análogamente, la instrucción

```
n = --i;
```

hace que el valor actual de `i` sea decrementado en 1 y se asigna entonces ese nuevo valor a `n`. En otras palabras, la instrucción es equivalente a

```
i = i - 1;
n = i;
```

En el Capítulo 3 (sección 3.2) se discuten con detalle expresiones como `++i` y `--i`.

Cuando se ejecuta el programa, se genera un diálogo interactivo como el que se muestra a continuación. De nuevo las respuestas del usuario están subrayadas.

```
Para PARAR, introducir 0 en el valor del radio

Radio = ? 3
Radio = ? 4
Radio = ? 5
Radio = ? 0
```

```
Relación de resultados

Radio = 3.000000    Area = 28.274309
Radio = 4.000000    Area = 50.265442
Radio = 5.000000    Area = 78.539749
```

Este sencillo programa no hace uso de los valores que se han almacenado en los arrays. Su único propósito es mostrar los mecanismos que utilizan los arrays. En un ejemplo más complicado, podríamos querer determinar el valor medio de las áreas, y entonces comparar cada área individualmente con la media. Para hacer esto tendríamos que recurrir a cada uno de los valores de las áreas (los elementos del array `area[0]`, `area[1]`, ..., etc.).

El uso de arrays se discute brevemente en el Capítulo 2 y extensamente en el capítulo 9.

EJEMPLO 1.13. Cálculo y almacenamiento de las áreas de varios círculos. Ésta es una aproximación más detallada al problema descrito en el ejemplo anterior.

```
/* programa para calcular áreas de varios círculos, usando
   un bucle while;
   los resultados se almacenan en un array;
   no se especifica el número de círculos;
   se introduce una cadena de caracteres para cada conjunto
   de datos */

#include <stdio.h>

#define PI 3.14159

float procesar(float radio); /* prototipo de función */

main()

{

  int n, i = 0;                /* declaración de variables */

  struct {
            char texto[20];
            float radio;
            float area;
  } circulo[10];               /* declaración de variable tipo
                                  estructura */

  printf("Para PARAR, introducir FIN en el identificador\n");
  printf("\nIdentificador: ");
  scanf("%s", circulo[i].texto);
  while (circulo[i].texto[0] !='F'
                                || circulo[i].texto[1] != 'I'
                                || circulo[i].texto[2] != 'N') {
         printf("Radio = ? ");
         scanf("%f", &circulo[i].radio);
```

```
            if (circulo[i].radio < 0)
                circulo[i].area = 0;
            else
                circulo[i].area = procesar(circulo[i].radio);

            ++i;
            printf("\nIdentificador: ");
                                        /* siguiente conjunto de datos */
            scanf("%s", circulo[i].texto);
    }

    n = --i;     /* El mayor valor de i */

    /* presentar los elementos del array */
    printf("\n\nRelación de resultados\n\n");
    for (i = 0; i <= n; ++i)
         printf("%s   Radio = %f   Area = %f\n",circulo[i].texto,
                                                 circulo[i].radio,
                                                 circulo[i].area);
}

float procesar(float r)      /* definición de función */

{

    float a;                 /* declaración de variable local */

    a = PI * r * r;
    return(a);
}
```

En este programa se introduce un texto *descriptor* , seguido del valor del radio, para cada círculo. Los caracteres del descriptor se almacenan en el array `texto`. Estos caracteres son en conjunto una *constante de cadena de caracteres* (ver sección 2.4). En este programa el tamaño máximo de cada constante de cadena de caracteres es 20.

El descriptor, el radio y el área correspondiente de cada círculo se definen como componentes de una *estructura* (ver Capítulo 11). Definimos entonces `circulo` como un array de estructuras. Es decir, cada elemento de `circulo` será una estructura que contiene un descriptor, un radio y un área. Por ejemplo, `circulo[0].texto` hace referencia al descriptor del primer círculo, `circulo[0].radio` hace referencia al radio del primer círculo y `circulo[0].area` hace referencia al área del primer círculo. (Hay que recordar que el sistema de numeración de los elementos del array comienza por 0, no por 1.)

Cuando se ejecuta el programa, se introduce un descriptor para cada círculo, seguido del valor del radio. Esta información se almacena en `circulo[i].texto` y `circulo[i].radio`. Se calcula el área correspondiente y se almacena en `circulo[i].area`. Este proceso continúa hasta que se introduce el descriptor `FIN`. Entonces se presenta toda la información almacenada en los elementos del array (descriptor, radio y área de cada círculo) y termina la ejecución.

Cuando se ejecuta el programa, se genera un diálogo interactivo como el que se muestra a continuación. Nótese que las respuestas del usuario están subrayadas de nuevo.

```
Para PARAR, introducir FIN en el identificador

Identificador: ROJO
Radio: 3

Identificador: BLANCO
Radio: 4

Identificador: AZUL
Radio: 5

Identificador: FIN


Relación de resultados

ROJO    Radio = 3.000000    Area = 28.274309
BLANCO    Radio = 4.000000    Area = 50.265442
AZUL    Radio = 5.000000    Area = 78.539749
```

1.7. CARACTERÍSTICAS DESEABLES DE UN PROGRAMA

Antes de concluir este capítulo examinemos brevemente algunas características importantes de los programas bien escritos. Estas características se pueden aplicar a programas no sólo escritos en C, sino en *cualquier* lenguaje de programación. Pueden aportarnos una serie de normas generales muy útiles para cuando comencemos a escribir próximamente, en este libro, nuestros propios programas en C.

1. *Integridad.* Se refiere a la corrección de los cálculos. Está claro que toda posible ampliación del programa no tendrá sentido si los cálculos no se realizan de forma correcta, pues la integridad de los cálculos es absolutamente necesaria en cualquier programa de computadora.
2. La *claridad* hace referencia a la facilidad de lectura del programa en conjunto, con particular énfasis en la lógica subyacente. Si un programa está escrito de forma clara, será posible para otro programador seguir la lógica del programa sin mucho esfuerzo. También hará posible al autor original seguir su propio programa después de haberlo dejado durante un período largo de tiempo. Uno de los objetivos al diseñar C fue el desarrollo de programas claros y de fácil lectura a través de un enfoque de la programación ordenado y disciplinado.
3. *Sencillez.* La claridad y correccción de un programa se suelen ver favorecidas con hacer las cosas de forma tan sencilla como sea posible, consistente con los objetivos del programa en su conjunto. De hecho puede ser deseable sacrificar cierta cantidad de eficiencia computacional con vistas a no complicar la estructura del programa.
4. La *eficiencia* está relacionada con la velocidad de ejecución y la utilización eficiente de la memoria. Éste es uno de los objetivos importantes, aunque no se debe conseguir a expensas de la pérdida de la claridad o la sencillez. Muchos programas complicados

conducen a un compromiso entre estas características. En tales situaciones es necesario recurrir a la experiencia y al sentido común.

5. *Modularidad.* Muchos programas se pueden dividir en pequeñas subtareas. Es una buena práctica de programación implementar cada una de estas subtareas como un módulo separado del programa. En C estos módulos son las funciones. El diseño modular de los programas aumenta la corrección y claridad de éstos y facilita los posibles cambios futuros del programa.
6. *Generalidad.* Normalmente queremos que un programa sea lo más general posible, dentro de unos límites razonables. Por ejemplo, podemos hacer un programa que lea los valores de ciertos parámetros en lugar de dejarlos fijos. Como norma general se puede conseguir con muy poco esfuerzo adicional un nivel considerable de generalidad.

CUESTIONES DE REPASO

1.1. ¿Qué es un «mainframe»? ¿Dónde se pueden encontrar? ¿Para qué se suelen utilizar normalmente?

1.2. ¿Qué es una computadora personal? ¿En qué se diferencian de los «mainframes»?

1.3. ¿Qué es una supercomputadora? ¿Y una minicomputadora? ¿Y una estación de trabajo? ¿En qué se diferencian unas de otras? ¿En qué difieren de los «mainframes» y de las computadoras personales?

1.4. Mencionar cuatro tipos de datos distintos.

1.5. ¿Qué se entiende por un programa de computadora? ¿Qué ocurre en general cuando se ejecuta un programa?

1.6. ¿Qué es la memoria de una computadora? ¿Qué clase de información se almacena en la memoria de una computadora?

1.7. ¿Qué es un bit? ¿Qué es un byte? ¿Cuál es la diferencia entre un byte y una palabra de memoria?

1.8. ¿Qué términos se utilizan para describir la memoria de una computadora? ¿Cuáles son los tamaños típicos de las memorias?

1.9. Mencionar algunos dispositivos auxiliares de almacenamiento. ¿En qué se diferencian estos dispositivos de almacenamiento de la memoria principal?

1.10. ¿Qué unidad de tiempo se utiliza para expresar la velocidad en la que se realizan las tareas elementales en una computadora?

1.11. ¿Cuál es la diferencia entre el procesamiento por lotes y el tiempo compartido? ¿Qué ventajas y desventajas presenta cada uno?

1.12. ¿Qué significa la computación interactiva? ¿Para qué tipo de sistemas se adecúan mejor estos sistemas?

1.13. ¿Qué es el lenguaje máquina? ¿En qué se diferencian el lenguaje máquina y los lenguajes de alto nivel?

1.14. Mencionar algunos lenguajes de alto nivel de uso frecuente. ¿Cuáles son las ventajas de utilizar lenguajes de alto nivel?

1.15. ¿Qué se entiende por compilación? ¿Qué significa interpretación? ¿En qué se diferencian estos dos procesos?

1.16. ¿Qué es un programa fuente? ¿Y un programa objeto? ¿Por qué son importantes estos conceptos?

1.17. ¿Cuáles son las características generales de C?

1.18. ¿Dónde y por quién fue desarrollado C? ¿Por qué se ha estandarizado este lenguaje?

1.19. ¿Qué es C++? ¿Qué relación hay entre C y C++?

1.20. ¿Cuáles son los componentes principales de un programa en C? ¿Qué significado lleva asociado el nombre `main`?

1.21. Describir la composición de una función en C.

1.22. ¿Qué son los argumentos? ¿Dónde aparecen los argumentos en un programa en C? ¿Qué otro término se utiliza a veces en lugar de argumento?

1.23. ¿Qué es una instrucción compuesta? ¿Cómo se escriben las instrucciones compuestas?

1.24. ¿Qué es una instrucción de expresión? ¿Se puede incluir una instrucción de expresión en una instrucción compuesta? ¿Se puede incluir una instrucción compuesta en una instrucción de expresión?

1.25. ¿Cómo se pueden incluir los comentarios en un programa en C? ¿Dónde se pueden poner los comentarios?

1.26. ¿Es necesario escribir los programas en C en minúsculas? ¿Se pueden utilizar para algo las mayúsculas en un programa en C? Explicarlo.

1.27. ¿Qué es una instrucción de asignación? ¿Cuál es la relación entre una instrucción de asignación y una instrucción de expresión?

1.28. ¿Qué signo de puntuación se pone al final de la mayoría de las instrucciones en C? ¿Terminan todas las instrucciones de esta forma?

1.29. ¿Por qué se sangran algunas de las instrucciones de un programa en C? ¿Por qué se incluyen normalmente líneas en blanco en un programa en C?

1.30. Comentar brevemente el significado de cada una de las siguientes características de los programas: integridad, claridad, sencillez, eficiencia, modularidad y generalidad. ¿Por qué es importante cada una de estas características?

PROBLEMAS

1.31. Determinar, lo mejor que se pueda, el propósito de cada uno de los siguientes programas en C. Identificar todas las variables de cada programa. Identificar todas las instrucciones de entrada y salida, todas las instrucciones de asignación y cualquier otra característica importante que se reconozca.

a)
```
main()

{

    printf("¡Bienvenido a la Informática !\n");
}
```

b)
```
#define MENSAJE "¡Bienvenido a la informática!"

main()

{

  printf(MENSAJE);

}
```

c)
```
main()

{

  float base, altura, area;

  printf("Base: ");
  scanf("%f",&base);
  printf("Altura: ");
  scanf("%f",&altura);
  area = (base * altura) / 2.;
  printf("Area: %f",area);

}
```

d)
```
main()

{

  float bruto, impuesto, neto;

  printf("Salario bruto: ");
  scanf("%f", &bruto);
  impuesto = 0.14 * bruto;
  neto = bruto - impuesto;
  printf("Impuestos: %.2f\n", impuesto);
  printf("Salario neto: %.2f", neto);

}
```

e)
```
int menor(int a, int b);

main()

{

  int a, b, min;

  printf("Introduzca el primer número: ");
  scanf("%d", &a);
```

```
    printf("Introduzca el segundo número: ");
    scanf("%d", &b);

    min = menor(a, b);

    printf("\nEl número menor es: %d", min);

}

int menor(int a, int b)

{

    if (a <= b)
         return(a);
    else
         return(b);

}
```

f)

```
int menor(int a, int b);

main()
{
    int cont, n, a, b, min;

    printf("¿Cuantos pares de números? ");
    scanf("%d", &n);

    for (cont = 1;cont <= n; ++cont) {
         printf("\nIntroduzca el primer número: ");
         scanf("%d", &a);
         printf("Introduzca el segundo número: ");
         scanf("%d", &b);

         min = menor(a, b);

         printf("\nEl número menor es: %d\n", min);
    }

}

int menor(int a, int b)

{

    if (a <= b)
     return(a);
    else
     return(b);

}
```

g)
```
int menor(int a, int b);

main()

{

  int a, b, min;

  printf("Para PARAR, introducir 0 en cada número\n");

  printf("\nIntroduzca el primer número: ");
  scanf("%d", &a);
  printf("Introduzca el segundo número: ");
  scanf("%d", &b);

  while (a != 0 || b != 0) {

    min = menor(a, b);
    printf("\nEl número menor es : %d\n", min);

    printf("\nIntroduzca el primer número: ");
    scanf("%d", &a);
    printf("Introduzca el segundo número: ");
    scanf("%d", &b);

  }

}

int menor(int a, int b)

{

  if (a <= b)
    return(a);
  else
    return(b);

}
```

h)
```
int menor(int a, int b);

main()

{

  int n, i = 0;
  int a[100], b[100], min[100];

  printf("Para PARAR, introducir 0 en cada número\n");

  printf("\nIntroduzca el primer número: ");
  scanf("%d", &a[i]);
```

```
      printf("Introduzca el segundo número: ");
      scanf("%d", &b[i]);

      while (a[i] || b[i]) {

         min[i] = menor(a[i], b[i]);

         printf("\nIntroduzca el primer número: ");
         scanf("%d", &a[++i]);
         printf("Introduzca el segundo número: ");
         scanf("%d", &b[i]);

      }

      n = -- i;

      printf("\nRelación de resultados\n\n");
      for (i = 0; i <= n; ++i)
         printf("a = %d   b = %d   min = %d\n", a[i], b[i], min[i]);

   }

int menor(int a, int b)

{

        if (a <= b)
            return(a);
        else
             return(b);

}
```

CAPÍTULO 2

Conceptos básicos de C

En este capítulo presentaremos los elementos básicos para la construcción de instrucciones simples de C; entre éstos se encuentra el conjunto de caracteres, los identificadores y palabras reservadas de C, los tipos de datos, las constantes, variables y arrays, las declaraciones, expresiones e instrucciones. Veremos cómo combinar estos elementos para formar componentes más grandes de programas.

Parte de este material se presenta de forma detallada y puede resultar difícil de asimilar, sobre todo para programadores con poca experiencia. Hay que señalar que el propósito de este capítulo es presentar ciertos conceptos básicos y algunas definiciones necesarias para los temas que se tratan en los siguientes capítulos. Por tanto, cuando lea este capítulo por primera vez puede ser suficiente adquirir una cierta familiaridad con los conceptos que se presentan. Se conseguirá una comprensión más profunda de estos elementos tras las repetidas referencias a este capítulo que se encuentran en los siguientes.

2.1. EL CONJUNTO DE CARACTERES DE C

Para formar los elementos básicos del programa (constantes, variables, operadores, expresiones, etc.), C utiliza como bloques de construcción las letras mayúsculas de la `A` a la `Z`, las minúsculas de la `a` a la `z`, los dígitos del `0` al `9` y ciertos caracteres especiales. Se presenta a continuación una lista de estos caracteres especiales:

<table>
<tr><td>+</td><td>-</td><td>*</td><td>/</td><td>=</td><td>%</td><td>&</td><td>#</td></tr>
<tr><td>!</td><td>?</td><td>^</td><td>"</td><td>'</td><td>~</td><td>\</td><td>|</td></tr>
<tr><td><</td><td>></td><td>(</td><td>)</td><td>[</td><td>]</td><td>{</td><td>}</td></tr>
<tr><td>:</td><td>;</td><td>.</td><td>,</td><td>_</td><td colspan="3">(espacio en blanco)</td></tr>
</table>

La mayoría de las versiones del lenguaje también permiten que otros caracteres, como `@` y `$`, se incluyan en cadenas de caracteres y comentarios.

C utiliza ciertas combinaciones de estos caracteres, como `\b`, `\n` y `\t`, para representar elementos especiales como el retroceso de un espacio, nueva línea y un tabulador, respectivamente. Estas combinaciones de caracteres se conocen como *secuencias de escape*. Trataremos las secuencias de escape en la sección 2.4. Por ahora nos limitaremos a decir que cada secuencia de escape representa un solo carácter, aun cuando se escriba con dos o más caracteres.

2.2. IDENTIFICADORES Y PALABRAS RESERVADAS

Los *identificadores* son nombres que se les da a varios elementos de un programa, como variables, funciones y formaciones. Un identificador está formado por letras y dígitos, en cualquier orden, excepto *el primer carácter, que debe ser una letra.* Se pueden utilizar mayúsculas y minúsculas, aunque es costumbre utilizar minúsculas para la mayoría de los identificadores. No se pueden intercambiar mayúsculas y minúsculas (esto es, una letra mayúscula no es equivalente a la correspondiente minúscula.) El carácter de subrayado (_) se puede incluir también, y es considerado como una letra. Se suele utilizar este carácter en medio de los identificadores. Un identificador también puede comenzar con un carácter de subrayado, aunque en la práctica no se suele hacer.

EJEMPLO 2.1. Los siguientes nombres son identificadores válidos.

```
x          y12     suma_1     _temperatura
nombres    area    porc_imp   TABLA
```

Los siguientes nombres no son identificadores válidos por las razones señaladas.

`4num`	el primer carácter debe ser una letra
`"x"`	caracteres ilegales (")
`orden-no`	carácter ilegal (-)
`indicador error`	carácter ilegal (espacio en blanco)

No hay límite para la longitud de los identificadores. Algunas implementaciones de C reconocen sólo los ocho primeros caracteres, aunque la mayoría de ellas reconocen más (típicamente, 31 caracteres). El resto de los caracteres son utilizados para la comodidad del programador.

EJEMPLO 2.2. Los identificadores `suma_de_valores` y `suma_de_variaciones` son válidos gramaticalmente. Sin embargo, algunos compiladores de C pueden no ser capaces de distinguirlos, ya que ambos tienen las mismas ocho primeras letras. De esta forma, en un mismo programa sólo se podrá utilizar uno de estos identificadores.

Como norma general, un identificador debe tener los suficientes caracteres para que su significado se reconozca fácilmente; por otra parte, se debe evitar un excesivo número de caracteres.

EJEMPLO 2.3. Se está escribiendo un programa en C para calcular el valor futuro de una inversión. Los identificadores `valor` y `valor_futuro` son nombres simbólicos apropiados. Sin embargo, `v` y `fv` serían demasiado cortos, ya que no queda claro el significado de estos identificadores. Por otra parte, un identificador como `valor_futuro_de_una_inversion` no será adecuado por ser demasiado largo e incómodo.

Hay ciertas palabras *reservadas* que tienen en C un significado predefinido estándar. Las palabras reservadas sólo se pueden utilizar para su propósito ya establecido; no se pueden utilizar como identificadores definidos por el programador.

Las palabras reservadas son:

```
auto        extern      sizeof
break       float       static
case        for         struct
char        goto        switch
const       if          typedef
continue    int         union
default     long        unsigned
do          register    void
double      return      volatile
else        short       while
enum        signed
```

Existen compiladores que incluyen todas o algunas de las siguientes palabras reservadas:

```
ada         far         near
asm         fortran     pascal
entry       huge
```

Algunos compiladores de C pueden reconocer otras palabras reservadas. Para obtener una lista completa de las mismas para un compilador determinado se debe consultar su manual de referencia.

Observe que todas las palabras reservadas están en minúsculas. Como los caracteres en minúsculas y mayúsculas no son equivalentes, se puede utilizar una palabra reservada escrita en mayúsculas como un identificador. Esto no se suele hacer normalmente, y se considera propio de un estilo de programación pobre.

2.3. TIPOS DE DATOS

C ofrece distintos tipos de datos, cada uno de los cuales se puede encontrar representado de forma diferente en la memoria de la computadora. A continuación se presenta una lista de los tipos de datos básicos. También se dan los requerimientos de memoria típicos. (El requerimiento de memoria para cada tipo de datos numéricos determinará el rango permisible de valores para ese tipo de datos. Hay que señalar que las necesidades de memoria para cada tipo de datos pueden variar de un compilador de C a otro.)

Tipo de datos	*Descripción*	*Requisito típico de memoria*
`int`	Cantidad entera	2 bytes o una palabra (varía de un compilador a otro)
`char`	Carácter	1 byte
`float`	Número en coma flotante (un número que incluye punto decimal y/o exponente)	1 palabra (4 bytes)

Tipo de datos	*Descripción*	*Requisito típico de memoria*
`double`	Número en coma flotante de doble precisión (más cifras significativas y mayor valor posible del exponente)	2 palabras (8 bytes)

Los compiladores de C diseñados para computadoras personales o minicomputadoras (computadoras con longitud de palabra menor que 32 bits) representan generalmente una palabra con 4 bytes (32 bits).

Algunos tipos básicos de datos se pueden ampliar utilizando los *cualificadores* de tipos de datos `short` (corto), `long` (largo), `signed` (con signo) y `unsigned` (sin signo). Por ejemplo, se pueden definir cantidades enteras como `short int`, `long int` o `unsigned int` (estos tipos de datos se suelen escribir simplemente `short`, `long` o `unsigned`, y se supone que son enteros). La interpretación de un tipo de datos entero con un cualificador delante puede variar de un compilador de C a otro, aunque existe una relación de sentido común. De esta forma un `short int` requerirá menos memoria o la misma que un `int` ordinario, pero nunca más. De igual forma, un `long int` puede requerir la misma cantidad de memoria o más que un `int` ordinario, pero nunca una cantidad de memoria menor.

Si `short int` e `int` requieren la misma memoria (por ejemplo 2 bytes), entonces `long int`, generalmente, ocupará el doble (por ejemplo 4 bytes). O si `int` y `long int` ocupan la misma memoria (por ejemplo 4 bytes), entonces `short int` ocupará la mitad de memoria (por ejemplo 2 bytes). Recuerde que estas especificaciones pueden variar de un compilador de C a otro.

Un `unsigned int` ocupa la misma memoria que un `int` ordinario. Sin embargo, en el caso de un `int` ordinario (o un `short int` o un `long int`), el bit del extremo izquierdo se reserva para el signo. En un `unsigned int`, todos los bits se utilizan para representar el valor numérico. De esta forma, un `unsigned int` puede llegar a almacenar un valor numérico aproximadamente el doble que un `int` ordinario (aunque, por supuesto, no puede almacenar valores negativos). Por ejemplo, si un `int` ordinario puede variar de `-32768` a `+32767` (esto es propio de un `int` de 2 bytes), entonces un `unsigned int` podrá tomar valores comprendidos entre `0` y `65535`. El cualificador `unsigned` se puede aplicar también a otros `ints` ya cualificados, por ejemplo `unsigned short int` o `unsigned long int`.

El tipo `char` se utiliza para representar caracteres individuales. Por tanto, el tipo `char` requerirá sólo un byte de memoria. Cada tipo `char` tiene una representación como entero equivalente, de esta forma un `char` es realmente una clase especial de entero corto (ver sección 2.4). En la mayoría de los compiladores, un dato tipo `char` podrá tomar valores de `0` a `255`. Algunos compiladores representan el tipo de datos `char` con un rango de valores de `-128` a `+127`. También se pueden utilizar datos `unsigned char` (con valores de `0` a `255`), o datos `signed char` (con valores de `-128` a `+127`).

Algunos compiladores permiten la aplicación del cualificador `long` a `float` o `double`, por ejemplo `long float` o `long double`. De todos modos, el significado de estos tipos de datos varía de un compilador de C a otro. Así `long float` puede ser equivalente a `double`. Además, `long double` puede ser equivalente a `double`, o puede hacer referencia a un tipo de datos de doble precisión «extra largo», que requiera más de dos palabras de memoria.

Posteriormente se introducirán en este libro dos tipos de datos adicionales, `void` y `enum` (`void` es tratado en la sección 7.2; `enum` se discute en la sección 14.1).

Cada identificador que representa un número o un carácter dentro de un programa en C debe estar asociado a uno de los tipos de datos básicos antes de que el identificador aparezca en una instrucción ejecutable. Esto se lleva a cabo mediante una *declaración de tipo*, como se describe en la sección 2.6.

2.4. CONSTANTES

C tiene cuatro tipos básicos de constantes: *constantes enteras, constantes en coma flotante, constantes de carácter* y *constantes de cadena de caracteres* (hay también *constantes enumeradas*, que se tratan en la sección 14.1). Es más, hay distintas clases de constantes enteras y en coma flotante, como se discute a continuación.

Las constantes enteras y en coma flotante representan números. Se las denomina, en general, constantes de *tipo numérico*. Las siguientes reglas se pueden aplicar a todas las constantes numéricas.

1. No se pueden incluir comas ni espacios en blanco en la constante.
2. Si se desea, la constante puede ir precedida de un signo menos (–). (Realmente, el signo menos es un *operador* que cambia el signo de una constante positiva, aunque se puede ver como parte de la constante misma.)
3. El valor de una constante no puede exceder un límite máximo y un mínimo especificados. Para cada tipo de constante, estos límites varían de un compilador de C a otro.

Veamos cada tipo de constante individualmente.

Constantes enteras

Una *constante entera* es un número con un valor entero, consistente en una secuencia de dígitos. Las constantes enteras se pueden escribir en tres sistemas numéricos diferentes: decimal (base 10), octal (base 8) y hexadecimal (base 16). Normalmente, los programadores que se están iniciando no utilizarán más que las constantes enteras decimales.

Una constante entera *decimal* puede ser cualquier combinación de dígitos tomados del conjunto de 0 a 9. Si la constante tiene dos o más dígitos, el primero de ellos debe ser distinto de 0.

EJEMPLO 2.4. A continuación se muestran varias constantes enteras decimales.

```
0    1    743    5280    32767    9999
```

Las siguientes constantes enteras decimales están escritas incorrectamente por las razones que se indican.

`12,245`	carácter ilegal (`,`).
`36.0`	carácter ilegal (`.`).
`10 20 30`	carácter ilegal (espacio en blanco).
`123-45-6789`	carácter ilegal (`-`).
`0900`	el primer dígito no puede ser cero.

Una constante entera *octal* puede estar formada por cualquier combinación de dígitos tomados del conjunto `0` a `7`. El primer dígito debe ser obligatoriamente `0`, con el fin de identificar la constante como un número octal.

EJEMPLO 2.5. A continuación se muestran varias constantes enteras octales.

```
0          01          0743          077777
```

Las siguientes constantes enteras octales están escritas de forma incorrecta por las razones que se señalan.

`743`	no comienza por `0`
`05280`	dígito ilegal (`8`)
`0777.777`	carácter ilegal (`.`)

Una constante entera *hexadecimal* debe comenzar por `0x` o `0X`. Puede aparecer después cualquier combinación de dígitos tomados del conjunto de `0` a `9` y de `a` a `f` (tanto minúsculas como mayúsculas). Las letras de la `a` a la `f` (o de la `A` a la `F`) representan las cantidades (decimales) 10 a 15, respectivamente.

EJEMPLO 2.6. A continuación se muestran varias constantes enteras hexadecimales.

```
0x          0X1          0X7FFF          0xabcd
```

Las siguientes constantes enteras hexadecimales están escritas de forma incorrecta por las razones que se señalan.

`0X12.34`	carácter ilegal (`.`)
`0BE38`	no comienza por `0x` o `0X`.
`0x.4bff`	carácter ilegal (`.`).
`0XDEFG`	carácter ilegal (`G`).

El valor de una constante entera se ha de encontrar entre cero y algún valor máximo que varía de una computadora a otra (y de un compilador a otro en una misma computadora). Un valor máximo típico en la mayoría de las computadoras personales y muchas minicomputadoras es `32767` en decimal (equivalente a `77777` en octal o a `7fff` en hexadecimal), o lo que es igual, $2^{15} - 1$. Las grandes computadoras («mainframes») suelen permitir valores más grandes, tales como `2 147 483 647` (esto es, $2^{31} - 1$) *. El lector debe determinar el valor de la versión de C que utilice en su computadora.

Constantes enteras largas y sin signo

Las constantes enteras *sin signo* pueden tener un valor máximo de aproximadamente el doble del máximo de las constantes enteras ordinarias, pero su valor no puede ser negativo *. Una cons-

* Supóngase que una computadora utiliza una palabra de n bits. Entonces, una cantidad entera ordinaria se encontrará en el rango -2^{n-1} a $+2^{n-1} - 1$, y una cantidad entera sin signo variará entre 0 y $2^n - 1$. Para un entero corto habrá que sustituir n por $n/2$, y para uno largo, n por $2n$. Estas reglas pueden variar de una computadora a otra.

tante entera sin signo se identifica añadiéndole la letra `U`, mayúscula o minúscula (`U` del inglés `unsigned`), al final de la constante.

Las constantes enteras *largas* pueden tomar valores máximos mayores que las constantes enteras ordinarias, pero ocupan más memoria de la computadora. En algunas computadoras (y/o algunos compiladores) se generará una constante entera larga cuando simplemente se especifique una cantidad que exceda el valor máximo. En cualquier caso, *siempre es posible* especificar una constante entera larga añadiendo la letra `L` (mayúscula o minúscula) al final de ésta.

Una constante entera larga sin signo se puede especificar añadiendo las letras `UL` al final de la constante. Las letras pueden estar en mayúsculas o minúscula. Sin embargo, la `U` debe ir delante de la `L`.

EJEMPLO 2.7. A continuación se muestran varias constantes enteras largas y sin signo.

Constante	*Sistema de numeración*
`50000U`	decimal (sin signo)
`123456789L`	decimal (larga)
`123456789UL`	decimal (larga sin signo)
`0123456L`	octal (larga)
`0777777U`	octal (sin signo)
`0X50000U`	hexadecimal (sin signo)
`0XFFFFFUL`	hexadecimal (larga sin signo)

Los valores máximos permitidos de las constantes enteras largas y sin signo varían de una computadora (y de un compilador) a otra. En algunas computadoras, el valor máximo permitido de una constante entera larga puede ser el mismo que el de una constante entera ordinaria; otras computadoras permiten que el valor de una constante entera larga sea mucho mayor que el de una ordinaria. Se recomienda de nuevo al lector que determine estos valores en su versión particular de C.

Constantes en coma flotante

Una *constante en coma flotante* es un número en base `10` que contiene un punto decimal o un exponente (o ambos).

EJEMPLO 2.8. A continuación se muestran varias constantes en coma flotante.

```
0.        1.          0.2         827.602
50000.    0.000743    12.3        315.0066
2E-8      0.006e-3    1.6667E+8   .12121212e12
```

Las siguientes *no* son constantes en coma flotante por las razones indicadas.

`1`	Deben encontrarse presentes un punto decimal o un exponente.
`1,000.0`	Carácter ilegal (,).
`2E+10.2`	El exponente debe ser una cantidad entera (no puede contener un punto decimal).
`3E 10`	Carácter ilegal (espacio en blanco) en el exponente.

Si existe un exponente, su efecto es el de desplazar la posición del punto decimal a la derecha si el exponente es positivo, o a la izquierda si es negativo. Si no incluye punto decimal en el número, se supone que se encuentra a la derecha del último dígito.

La interpretación de una constante en coma flotante con exponente es justamente la misma que en notación científica, excepto que se sustituye la base `10` por la letra `E` (o `e`). De esta forma, el número 1.2×10^{-3} se debería escribir como `1.2E-3` o `1.2e-3`. Esto es equivalente a `0.12e-2` o `12e-4`, etc.

EJEMPLO 2.9. La cantidad 3×10^5 se puede representar en C por cualquiera de las siguientes constantes en coma flotante:

```
300000.    3e5     3e+5    3E5      3.0e+5
.3e6       0.3E6   30E4    30.E+4   300e3
```

De igual forma, la cantidad 5.026×10^{-17} se puede representar con cualquiera de las siguientes constantes en coma flotante:

```
5.026E-17    0.5026e-16    50.26e-18    0.0005026E-13
```

Los valores que pueden tener las constantes en coma flotante se encuentran dentro de un rango mucho mayor que el de las constantes enteras. Típicamente, la magnitud de una constante en coma flotante puede variar entre un valor mínimo de aproximadamente `3.4E-38` y un máximo de `3.4E+38`. Algunas versiones del lenguaje permiten constantes en coma flotante que cubren un rango mucho mayor, como de `1.7E-308` a `1.7E+308`. También es una constante en coma flotante válida el valor `0.0` (que es menor aún que `3.4E-38` o `1.7E-308`). El lector debe ocuparse de averiguar estos valores para su computadora y su versión particular de C.

Las constantes en coma flotante se representan en C normalmente como cantidades de doble precisión. Por tanto, cada constante en coma flotante ocupará, típicamente, dos palabras (8 bytes) de memoria. Algunas versiones de C permiten la especificación de constantes en coma flotante de «simple precisión», añadiendo la letra `F` (mayúscula o minúscula) al final de la constante (por ejemplo `3E5F`). De forma análoga, algunas versiones de C permiten la especificación de una constante en coma flotante «larga» añadiendo la letra `l` (mayúscula o minúscula) al final de la constante (por ejemplo `0.123456789E-33L`).

La precisión de las constantes en coma flotante (el número de cifras significativas) puede variar de una versión de C a otra. De hecho, todas las versiones del lenguaje permiten al menos seis cifras significativas, y algunas versiones proporcionan dieciocho cifras significativas. De nuevo el lector debe ocuparse de determinar ese valor para su versión de C.

Precisión numérica

Debe quedar claro que las constantes enteras son cantidades exactas, mientras que las constantes en coma flotante son aproximaciones. Las razones de esto se encuentran fuera del ámbito de esta discusión. En cualquier caso, el lector debe tener presente que la constante en coma flotante `1.0` puede ser representada en la memoria de la computadora como `0.99999999...`, aun cuando aparezca como `1.0` cuando se presente por pantalla (debido al redondeo automático). Por esta razón no se pueden utilizar los valores en coma flotante para ciertas funciones, tales como conteo, indexación, etc., en las que son necesarios valores exactos. Discutiremos estas restricciones según vayan apareciendo en próximos capítulos de este libro.

Constantes de carácter

Una *constante de carácter* es un solo carácter, encerrado con comillas simples.

EJEMPLO 2.10. A continuación se muestran varias constantes de carácter.

```
'A'     'X'     '3'     '?'     ' '
```

Observe que la última constante consiste en un espacio en blanco encerrado con comillas simples.

Las constantes de carácter tienen valores enteros determinados por el conjunto de caracteres particular de la computadora. Por tanto, el valor de una constante de carácter puede variar de una computadora a otra. Sin embargo, las constantes en sí son independientes del conjunto de caracteres. Este hecho elimina la dependencia de un programa en C de un conjunto de caracteres en particular (más sobre esto más adelante).

La mayoría de las computadoras, y prácticamente todas las computadoras personales, utilizan el conjunto de caracteres ASCII (Código Estándar Americano para el Intercambio de Información), en el cual cada carácter individual se codifica numéricamente con su propia combinación única de 7 bits (existen, pues, un total de $2^7 = 128$ caracteres diferentes). La Tabla 2.1 contiene el conjunto de caracteres ASCII, donde aparece también el equivalente en decimal de los 7 bits que representan a cada carácter. Oberve que además de codificados, los caracteres están ordenados. En particular, los dígitos están ordenados consecutivamente en su propia secuencia numérica (0 a 9), y las letras están dispuestas en orden alfabético, precediendo las mayúsculas a las minúsculas. Esto permite que las unidades de datos de tipo carácter se puedan comparar entre sí, basándose en su orden relativo dentro del conjunto de caracteres.

EJEMPLO 2.11. A continuación se muestran varias constantes de carácter y sus correspondientes valores en el conjunto de caracteres ASCII.

Constante	*Valor*
`'A'`	`65`
`'x'`	`120`
`'3'`	`51`
`'?'`	`63`
`' '`	`32`

Tabla 2.1. El conjunto de caracteres ASCII

Valor ASCII	Carácter	Valor ASCII	Carácter	Valor ASCII	Carácter	Valor ASCII	Carácter
0	NUL	32	espacio en blanco	64	@	96	`
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	'	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	\	124	|
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	DEL

Los primeros 32 caracteres y el último son caracteres de control. Normalmente no se pueden presentar. Sin embargo, algunas versiones de C (en algunas computadoras) soportan caracteres gráficos especiales para dichos valores ASCII. Por ejemplo, 001 puede representar el carácter ☺, 002 puede representar ☻, y así sucesivamente.

Estos valores serán los mismos para todas las computadoras que utilicen el conjunto de caracteres ASCII. Sin embargo, serán diferentes para computadoras que utilicen un conjunto de caracteres alternativo.

Los «mainframe» de IBM utilizan el conjunto de caracteres EBCDIC (Código de Información Decimal Codificada en Binario Extendido), en el cual cada carácter individual se codifica numéricamente con una combinación única de 8 bits. El conjunto de caracteres EBCDIC es diferente del conjunto ASCII.

Secuencias de escape

Ciertos caracteres no imprimibles, así como la barra inclinada hacia atrás (\) y la comilla simple ('), se pueden expresar en términos de *secuencias de escape*. Una secuencia de escape siempre comienza con una barra inclinada hacia atrás y es seguida por uno o más caracteres especiales. Por ejemplo, un salto de línea (`LF`), que se denomina *carácter de nueva línea* en C, se puede representar como `\n`. Una secuencia de escape siempre representa un solo carácter, aun cuando se escriba con dos o más caracteres.

A continuación se listan las secuencias de escape utilizadas con mayor frecuencia.

Carácter	Secuencia de escape	Valor ASCII
sonido (alerta)	`\a`	007
retroceso	`\b`	008
tabulador horizontal	`\t`	009
tabulador vertical	`\v`	011
nueva línea (avance de línea)	`\n`	010
avance de página	`\f`	012
retorno de carro	`\r`	013
comillas (")	`\"`	034
comilla simple (')	`\'`	039
signo de interrogación (?)	`\?`	063
barra inclinada hacia atrás (\)	`\\`	092
nulo	`\0`	000

EJEMPLO 2.12. A continuación se muestran varias constantes de carácter, expresadas como secuencias de escape.

```
'\n'     '\t'     '\b'     '\''     '\\'     '\"'
```

Observe que las tres últimas secuencias de escape representan una comilla simple, una barra inclinada hacia atrás y unas comillas dobles, respectivamente.

La secuencia de escape `\0` es de especial interés. Representa el *carácter nulo* (ASCII 000), que se utiliza para indicar el final de una *cadena de caracteres* (ver más adelante). Hay que señalar que la constante de carácter nulo `'\0'` no es equivalente a la constante de carácter `'0'`.

Una secuencia de escape también se puede expresar en términos de uno, dos o tres dígitos octales que representan patrones de bits correspondientes a un carácter. La forma general de tal secuencia de escape es `\ooo`, donde cada `o` representa un dígito octal (de `0` a `7`). Algunas ver-

siones de C también permiten expresar una secuencia de escape en términos de uno o más dígitos hexadecimales, precedidos por la letra x. La forma general de una secuencia de escape en hexadecimal es \xhh, donde cada h representa un dígito hexadecimal (de 0 a 9 y de a a f). Las letras pueden estar tanto en mayúsculas como en minúsculas. Utilizar una secuencia de escape en octal o en hexadecimal es normalmente menos recomendable que escribir directamente la constante de carácter, ya que los patrones de bits dependen de los conjuntos de caracteres particulares.

EJEMPLO 2.13. La letra A se representa por el valor decimal 065 en el conjunto de caracteres ASCII. Este valor es equivalente al octal 101. (El valor binario equivalente es 001 000 001.) Por tanto, la constante de carácter 'A' se puede expresar como la secuencia de escape en octal '\101'.

En algunas versiones de C, la letra A también se puede expresar como una secuencia de escape en hexadecimal. El equivalente en hexadecimal al valor decimal 65 es 41. (El equivalente en binario es 0100 0001.) Por tanto, la constante de carácter 'A' se puede expresar como '\x41' o como '\X41'.

La mejor forma de representar esta constante de carácter es simplemente 'A'. De esta forma, la constante de carácter no depende de su representación en ASCII.

Las secuencias de escape sólo se pueden escribir para ciertos caracteres especiales, tales como los listados antes, o en términos de dígitos octales o hexadecimales. Si una barra inclinada hacia atrás es seguida por cualquier otro carácter, el resultado es impredecible. Se ignorará simplemente en la mayoría de los casos.

Constantes de cadena de caracteres

Una *constante de cadena de caracteres* consta de cualquier número de caracteres consecutivos (o ninguno), encerrados entre comillas (dobles).

EJEMPLO 2.14. A continuación se muestran varias constantes de cadena de caracteres.

```
"verde"      "Washinton, D. C. 20005"      "270-32-3456"
"$19.95"     "LA RESPUESTA CORRECTA ES"    "2*(I+3)/J"
"     "      "Línea 1\nLínea 2\nLínea 3"   ""
```

Nótese que la constante de cadena de caracteres "Línea 1\nLínea 2\nLínea 3" ocupa tres líneas, a causa de los caracteres de nueva línea que se incluyen en ella. Esta constante de cadena de caracteres se mostraría así:

```
Línea 1
Línea 2
Línea 3
```

Advertir también que "" es una cadena de caracteres *nula* (vacía).

A veces es necesario incluir ciertos caracteres especiales (como la barra inclinada hacia atrás o las comillas) en una constante de cadena de caracteres. Estos caracteres se *deben* representar en términos de sus secuencias de escape. De igual forma, ciertos caracteres no imprimibles (por ejemplo el tabulador, nueva línea) se pueden incluir en cadenas de caracteres si se representan en términos de sus correspondientes secuencias de escape.

EJEMPLO 2.15. La siguiente constante de cadena de caracteres incluye tres caracteres especiales que se representan por sus correspondientes secuencias de escape.

```
"\tPara continuar, pulsar la tecla \"RETURN\"\n"
```

Los caracteres especiales son `\t` (tabulador horizontal), `\"` (comillas, que aparece dos veces) y `\n` (nueva línea).

El compilador inserta automáticamente un carácter nulo (`\0`) al final de toda constante de cadena de caracteres, como el último carácter de ésta (antes de finalizar con las comillas dobles). Este carácter no aparece cuando se visualiza la cadena. Sin embargo, podemos examinar individualmente los caracteres de una cadena de forma fácil y comprobar si cada uno de ellos es o no un carácter nulo (veremos cómo hacer esto en el Capítulo 6). De esta forma se puede identificar rápidamente el final de una cadena de caracteres. Esto es de gran ayuda si la cadena es examinada carácter a carácter, como se requiere en muchas aplicaciones. Además, en muchas situaciones esta designación del final de la cadena elimina la necesidad de especificar una longitud máxima de las cadenas de caracteres.

EJEMPLO 2.16. La constante de cadena de caracteres que aparece en el Ejemplo 2.15 contiene realmente 43 caracteres. Incluye cinco espacios en blanco, cuatro caracteres especiales (tabulador horizontal, dos comillas dobles y uno de nueva línea) representados por secuencias de escape y el carácter nulo (`\0`) al final de la cadena.

Recordar aquí que una constante de carácter (por ejemplo `'A'`) y su correspondiente constante de cadena de caracteres de uno sólo (`"A"`) no son equivalentes. Señalar también que una constante de carácter tiene un valor entero correspondiente, mientras que una constante de cadena de un solo carácter no tiene un valor entero equivalente y de hecho consta de *dos* caracteres —el carácter especificado seguido de un carácter nulo (`\0`).

EJEMPLO 2.17. La constante de carácter `'w'` tiene un valor entero de `119` en el conjunto de caracteres ASCII. No tiene un carácter nulo al final. Sin embargo, la constante de cadena de caracteres `"w"` consiste en realidad en dos caracteres, la letra minúscula `w` y el carácter nulo `\0`. Esta constante no tiene un valor entero correspondiente.

2.5. VARIABLES Y ARRAYS

Una *variable* es un identificador que se utiliza para representar cierto tipo de información dentro de una determinada parte del programa. En su forma más sencilla, una variable es un identificador que se utiliza para representar un dato individual; es decir, una cantidad numérica o una constante de carácter. En alguna parte del programa se asigna el dato a la variable. Este valor se puede recuperar después en el programa con simplemente hacer referencia al nombre de la variable.

A una variable se le pueden asignar diferentes valores en distintas partes del programa. De esta forma la información representada puede cambiar durante la ejecución del programa. Sin embargo, el tipo de datos asociado a la variable no puede cambiar.

EJEMPLO 2.18. Un programa en C contiene las siguientes líneas:

```
int a, b, c;
char d;

. . .

a = 3;
b = 5;
c = a + b;
d = 'a';

. . .

a = 4;
b = 2;
c = a - b;
d = 'W';
```

Las dos primeras líneas son *declaraciones de tipo*, en las cuales se establece que a, b y c son variables enteras y que d es una variable de tipo carácter. De esta forma a, b y c representarán sendas cantidades enteras y d representará un carácter. Estas declaraciones de tipo se mantienen para todo el programa (más sobre esto en la sección 2.6).

Las siguientes cuatro líneas hacen lo siguiente: a a se le asigna la cantidad entera 3, a b se le asigna 5 y a c se le asigna la suma de a + b (es decir 8). A d se le asigna el carácter 'a'.

En la tercera línea de este grupo puede verse cómo se accede a los valores de las variables a y b simplemente escribiéndolas a la derecha del signo igual.

Las últimas cuatro líneas cambian los valores asignados a las variables de la forma siguiente: la cantidad entera 4 es asignada a a, sustituyendo el anterior valor 3; después se asigna 2 a b, reemplazando al valor anterior, 5; se le asigna a c la diferencia entre a y b (es decir, 2), reemplazando al anterior valor, 8. Finalmente, se le asigna a d el carácter 'W', sustituyendo al anterior carácter, 'a'.

El *array* es otra clase de variable que se utiliza con frecuencia en C. Un array es un identificador que referencia una *colección* de datos con el mismo nombre. Los datos deben ser del mismo tipo (por ejemplo, todos enteros, todos caracteres, etc.). Cada uno de estos datos es representado por su *elemento del array* correspondiente (por ejemplo, el primer dato es representado por el primer elemento del array, etc.). Los elementos individuales del array se distinguen unos de otros por el valor que se le asigna al *índice*.

EJEMPLO 2.19. Supongamos que x es un array de 10 elementos. El primer elemento es x[0], el segundo x[1], y así sucesivamente. El último elemento será x[9].

El índice asociado a cada elemento se encierra entre corchetes. De esta forma, el valor del índice para acceder al primer elemento es 0, para acceder al segundo elemento es 1, y así sucesivamente. Para un array de n elementos, los valores del índice se encontrarán entre 0 y n-1.

Hay diferentes tipos de arrays (arrays de enteros, arrays de caracteres, arrays unidimensionales, arrays multidimensionales). Por ahora concentraremos nuestra atención en un solo tipo de array: el array de caracteres unidimensional (también llamado array de tipo carácter). Generalmente se utiliza este tipo de array para representar una cadena de caracteres. Cada elemento del array representará un carácter de la cadena. De esta forma se puede ver al array en conjunto como una lista ordenada de caracteres.

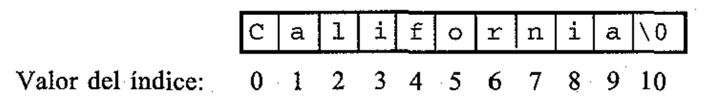
Como el array es unidimensional, tendrá un solo *índice*, cuyo valor determina los elementos individuales del array. Si el array contiene *n* elementos, el índice será una cantidad entera cuyos valores se encuentran entre `0` y `n-1`. Observe que una cadena de `n` caracteres requerirá un array de `(n+1)` elementos, debido al carácter nulo `(\0)` que se añade automáticamente al final de la cadena.

EJEMPLO 2.20. Supongamos que deseamos almacenar la cadena `"California"` en un array de caracteres unidimensional llamado `letras`. Como `"California"` contiene 10 caracteres, `letras` será un array de `11` elementos. De esta forma, `letras[0]` representará la letra `C`, `letras[1]` representará la letra `a`, y así sucesivamente, tal y como se representa a continuación. Observe que el último elemento, `letras[10]`, representa al carácter nulo que indica el final de la cadena.

Número de elemento	*Valor del índice*	*Elemento del array*	*Dato correspondiente (carácter de la cadena)*
1	0	letras[0]	C
2	1	letras[1]	a
3	2	letras[2]	l
4	3	letras[3]	i
5	4	letras[4]	f
6	5	letras[5]	o
7	6	letras[6]	r
8	7	letras[7]	n
9	8	letras[8]	i
10	9	letras[9]	a
11	10	letras[10]	\0

En esta lista se puede observar, por ejemplo, que el quinto elemento del array, `letras[4]`, representa la letra `f`, y así sucesivamente.

Los elementos del array y su contenido se muestran esquemáticamente en la Figura 2.1.



Un array de caracteres de once elementos

Figura 2.1.

Discutiremos los arrays con mayor detalle en los Capítulos 9 y 10.

2.6. DECLARACIONES

Una *declaración* asocia un tipo de datos especificado a un grupo de variables. Se deben declarar todas las variables antes de que aparezcan en las instrucciones ejecutables.

Una declaración consta de un tipo de datos, seguido de uno o más nombres de variables, finalizando con un punto y coma. (Los tipos de datos permitidos se han discutido en la sec-

ción 2.3.) Cada variable array debe ir seguida de un par de corchetes, con un entero positivo dentro de éstos que especifica el tamaño (el número de elementos) del array.

EJEMPLO 2.21. Un programa en C contiene las siguientes declaraciones de tipos:

```
int a, b, c;
float raiz1, raiz2;
char indicador, texto[80];
```

De esta forma se declaran `a`, `b` y `c` como variables enteras, `raiz1` y `raiz2` son variables en coma flotante, `indicador` una variable de tipo carácter y `texto` un array de tipo carácter de `80` elementos. Observe los corchetes que delimitan la especificación de tamaño de `texto`.

También se podrían haber escrito las declaraciones anteriores como sigue:

```
int a;
int b;
int c;
float raiz1;
float raiz2;
char indicador;
char texto[80];
```

Esta forma puede ser útil si se acompaña a cada variable con un comentario que explique su propósito. Sin embargo, en pequeños programas las variables del mismo tipo se suelen incluir en una sola declaración.

Las variables de tipo entero se pueden declarar como *enteros cortos* para pequeñas cantidades enteras, o *enteros largos* para cantidades enteras mayores. (Algunos compiladores de C se ocupan de reservar menos espacio de memoria para enteros cortos que para largos.) Estas variables se pueden declarar escribiendo `short int` y `long int`, o simplemente `short` y `long`, respectivamente.

EJEMPLO 2.22. Un programa en C contiene las siguientes declaraciones de tipo:

```
short int a, b, c;
long  int r, s, t;
int   p,  q;
```

Algunos compiladores de C reservarán menos espacio de memoria para las variables enteras cortas `a`, `b` y `c` que para las variables enteras `p` y `q`. Los valores típicos son dos bytes para cada variable declarada como entera corta y cuatro bytes (una palabra) para cada variable entera. Los valores máximos permisibles para `a`, `b` y `c` serán más pequeños que los de `p` y `q` cuando se utilice un compilador de este tipo.

De forma semejante, algunos compiladores reservarán espacio de memoria adicional para las variables enteras largas `r`, `s` y `t` que para las variables enteras `p` y `q`. Es un valor típico el de dos palabras (8 bytes) para cada variable entera larga y de una palabra (4 bytes) para cada variable entera ordinaria. Los valores máximos permisibles para `r`, `s` y `t` serán mayores que los de `p` y `q` si se utiliza un compilador de este tipo.

Las declaraciones anteriores se pueden escribir también como

```
short a, b, c;
long  r, s, t;
int   p, q;
```

Por consiguiente, `short` y `short int` son equivalentes, así como `long` y `long int`.

También se puede declarar una variable entera como *sin signo*, escribiendo `unsigned int`, o simplemente `unsigned`, como indicador de tipo. Las cantidades enteras sin signo pueden ser mayores que las cantidades enteras ordinarias (aproximadamente el doble), pero no pueden ser negativas.

EJEMPLO 2.23. Un programa en C contiene las siguientes declaraciones:

```
int a, b;
unsigned x, y;
```

Las variables sin signo `x` e `y` pueden representar valores aproximadamente dos veces mayores que los que pueden representar `a` y `b`. Sin embargo, `x` e `y` no pueden representar cantidades negativas. Por ejemplo, si la computadora utiliza 2 bytes para cada entero, `a` y `b` podrán tomar valores entre `-32768` y `+32767`, mientras que los valores de `x` e `y` podrán variar entre `0` y `+65535`.

Se pueden declarar variables en coma flotante como de *doble precisión* utilizando el indicador de tipo `double` o `long float` en lugar de `float`. En la mayoría de las versiones de C, el exponente de una cantidad de doble precisión es mayor en valor absoluto que el exponente de una cantidad ordinaria en coma flotante. Por tanto, la cantidad representada por una variable de doble precisión se podrá encontrar dentro de un rango mayor. Además, una cantidad de doble precisión tendrá mayor número de cifras significativas.

EJEMPLO 2.24. Un programa en C contiene las siguientes declaraciones:

```
float c1, c2, c3;
double raiz1, raiz2;
```

En un compilador de C en concreto, las variables de doble precisión `raiz1` y `raiz2` representan valores que pueden variar (en valor absoluto) aproximadamente entre $1{,}7 \times 10^{-308}$ y $1{,}7 \times 10^{+308}$. Sin embargo, las variables en coma flotante `c1`, `c2` y `c3` tienen como límites (en valor absoluto) $3{,}4 \times 10^{-38}$ y $3{,}4 \times 10^{+38}$. Además, los valores representados por `raiz1` y `raiz2` se almacenarán con 18 cifras significativas, mientras que los valores representados por `c1`, `c2` y `c3` lo estarán con sólo 6 cifras significativas.

La última declaración se podría haber escrito

```
long float raiz1, raiz2;
```

aunque la forma original (`double raiz1, raiz2;`) es más común.

Se le pueden asignar *valores iniciales* a las variables dentro de la declaración de tipo. Para hacer esto, la declaración debe consistir en un tipo de datos, seguido por un nombre de variable,

un signo igual (=) y una constante del tipo apropiado. Al final se debe poner, como de costumbre, un punto y coma (;).

EJEMPLO 2.25. Un programa en C contiene las siguientes declaraciones de tipo:

```
int    c = 12;
char   estrella = '*';
float  suma = 0.;
double factor = 0.21023e-6;
```

Por consiguiente, c es una variable entera cuyo valor inicial es 12, estrella es una variable de tipo carácter a la que se le asigna inicialmente el carácter '*', suma una variable en coma flotante cuyo valor inicial es 0. y factor una variable de doble precisión cuyo valor inicial es $0{,}21023 \times 10^{-6}$.

Un array de tipo carácter también se puede inicializar en una declaración. Para hacer esto, se suele escribir el array sin una especificación de tamaño explícita (los corchetes están vacíos). A continuación del nombre del array se escribe un signo igual, la cadena de caracteres (encerrada en comillas dobles) y un punto y coma. Ésta es una forma muy cómoda de asignar una cadena de caracteres a una formación de tipo carácter.

EJEMPLO 2.26. Un programa en C contiene la siguiente declaración de tipo:

```
char text[] = "California";
```

Esta declaración hará que texto sea un array de caracteres de 11 elementos. Los primeros 10 elementos representarán los diez caracteres de la palabra California, y el elemento undécimo representará el carácter nulo (\0) que se añade automáticamente al final de la cadena.

También se podría haber escrito la declaración de la siguiente forma:

```
char texto[11] = "California";
```

en donde el tamaño del array se especifica explícitamente. En estos casos es importante que el tamaño se especifique correctamente. Si se especifica un tamaño más pequeño que el necesario, por ejemplo,

```
char texto[10] = "California";
```

se perderán los caracteres del final de la cadena (en este caso, el carácter nulo). Si se especifica un tamaño demasiado grande, por ejemplo,

```
char texto[20] = "California";
```

se les asignarán ceros a los elementos sobrantes del array, o es posible que se rellenen con caracteres sin sentido.

Las declaraciones de arrays que incluyen asignaciones de valores iniciales sólo pueden aparecer en ciertas partes de un programa en C (ver Capítulo 9).

En el Capítulo 8 veremos que se puede clasificar a las variables por su *tipo de almacenamiento* lo mismo que por su tipo de datos. El tipo de almacenamiento especifica la parte del programa dentro de la cual se reconoce a la variable. Es más, el tipo de almacenamiento asociado a un array determina si se puede inicializar o no. Esto se explica en el Capítulo 9.

2.7. EXPRESIONES

Una *expresión* representa una unidad de datos simple, tal como un número o un carácter. La expresión puede consistir en una entidad simple, como una constante, una variable, un elemento de un array o una referencia a una función. También puede consistir en alguna combinación de tales entidades interconectadas por uno o más *operadores*. El uso de expresiones involucrando operadores es especialmente frecuente en C, como en muchos otros lenguajes de programación.

Las expresiones también pueden representar condiciones lógicas que son verdaderas o falsas. En C las condiciones *verdadero* y *falso* se representan por los valores `1` y `0`, respectivamente. Por tanto, las expresiones lógicas representan en realidad cantidades numéricas.

EJEMPLO 2.27. A continuación se muestran una serie de expresiones sencillas.

```
a + b
x = y
c = a + b
x <= y
x == y
++i
```

En la primera expresión aparece el *operador suma* `(+)`. La expresión representa la suma de los valores asignados a las variables `a` y `b`.

En la segunda expresión aparece el *operador de asignación* `(=)`. En este caso la expresión hace que el valor que contenga `y` se le asigne a `x`. Ya hemos visto en anteriores ejemplos el uso de este operador (ver ejemplos 1.6 a 1.13, 2.25 y 2.26). C posee varios operadores de asignación adicionales, como se discute en la sección 3.4.

En la tercera línea se asigna a la variable `c` el valor de la expresión `(a + b)`. Observe que se han combinado las características de las dos primeras expresiones (suma y asignación).

La cuarta expresión tendrá el valor `1` (verdadero) si el valor de `x` es menor o igual al de `y`. De otra forma, la expresión tendrá el valor `0` (falso). En esta expresión, <= es un *operador relacional* que compara los valores de las variables `x` e `y`.

La quinta expresión es una comprobación de igualdad (compárese con la segunda expresión, que es una expresión de asignación). Por consiguiente, la expresión tendrá el valor `1` (verdadero) si el valor de `x` es igual al valor de `y`. En cualquier otro caso, la expresión tendrá el valor `0` (falso).

La última expresión hace que el valor de la variable `i` se incremente en `1`. Por tanto, la expresión es equivalente a

```
i = i + 1;
```

El operador ++, que indica incremento en 1, es un operador *unario* porque sólo tiene un *operando* (en este caso, la variable `i`). C tiene varios operadores más de este tipo, como se discute en la sección 3.2.

El lenguaje C incluye muchas clases diferentes de operadores y expresiones. La mayoría se describen con detalle en el Capítulo 3. Otros se tratarán en cualquier otro lugar de este libro, según se vaya haciendo necesario.

2.8. INSTRUCCIONES

Una *instrucción* hace que la computadora efectúe alguna acción. Hay tres tipo diferentes de instrucciones en C. Éstas son las *instrucciones de expresión*, *instrucciones compuestas* e *instrucciones de control*.

Una instrucción de expresión consiste en una expresión seguida de un punto y coma. La ejecución de una instrucción de expresión hace que se evalúe la expresión.

EJEMPLO 2.28. A continuación se muestran varias instrucciones de expresión.

```
a = 3;
c = a + b;
++i;
printf("Area = %f",area);
;
```

Las dos primeras instrucciones de expresión son instrucciones de tipo asignación. Cada una hace que el valor de la expresión a la derecha del signo igual le sea asignado a la variable de la izquierda. La tercera instrucción de expresión es una instrucción de tipo incremento, que hace que el valor de `i` sea incrementado en `1`.

La cuarta instrucción de expresión hace que la función `printf` sea evaluada. Ésta es una función de biblioteca de C estándar que visualiza resultados en la computadora (más detalles en la sección 3.6). En este caso se visualizará el mensaje `Area =`, seguido del valor actual de la variable `area`. Así, si `area` tiene el valor `100.`, la instrucción generará el mensaje

```
Area = 100.000000
```

La última instrucción de expresión no hace nada, ya que consta sólo de un punto y coma. Es un sencillo mecanismo de conseguir una instrucción de expresión vacía en lugares donde se requiera este tipo de instrucción. Consecuentemente, se denomina *instrucción nula*.

Una instrucción compuesta está formada por varias instrucciones individuales encerradas con un par de llaves { }. Las instrucciones individuales pueden ser a su vez instrucciones de expresión, instrucciones compuestas o instrucciones de control. Por tanto, la instrucción compuesta hace posible incluir instrucciones dentro de otras instrucciones. A diferencia de una instrucción de expresión, una instrucción compuesta no acaba con un punto y coma.

EJEMPLO 2.29. Se muestra a continuación una instrucción compuesta.

```
{
  pi = 3.141593;
  circunferencia = 2 * pi * radio;
  area = pi * radio * radio;
}
```

Esta secuencia compuesta en particular consta de tres instrucciones de expresión de tipo asignación, aunque es considerada como una única entidad dentro del programa en que aparece. Observe que la instrucción compuesta no acaba con un punto y coma después de la llave.

Las instrucciones de control se utilizan para conseguir ciertas acciones especiales en los programas, tales como comprobaciones lógicas, bucles y bifurcaciones. Muchas instrucciones de control requieren que otras instrucciones se encuentren incluidas en ellas, como se ilustra en el siguiente ejemplo.

EJEMPLO 2.30. La siguiente instrucción de control crea un bucle condicional en el que se ejecutan varias acciones repetidamente, hasta que se satisface una condición en particular.

```
while (cont <= n) {
   printf("x = ");
   scanf("%f", &x);
   suma += x;
   ++cont;
}
```

Esta instrucción contiene una instrucción compuesta, que a su vez contiene cuatro instrucciones de expresión. La instrucción compuesta se seguirá ejecutando hasta que el valor de `cont` supere el valor de `n`. Nótese que `cont` se incrementa en uno en cada pasada por el bucle.

En el Capítulo 6 se tratan con mayor detalle las instrucciones de control.

2.9. CONSTANTES SIMBÓLICAS

Una *constante simbólica* es un nombre que sustituye una secuencia de caracteres. Los caracteres pueden representar una constante numérica, una constante de carácter o una constante de cadena de caracteres. Por tanto, una constante simbólica permite que aparezca un nombre en lugar de una constante numérica, una constante de carácter o una constante de cadena de caracteres. Cuando se compila un programa, cada aparición de una constante simbólica es reemplazada por su correspondiente secuencia de caracteres.

Las constantes simbólicas se suelen definir al comienzo del programa. Las constantes simbólicas pueden entonces aparecer después en el programa en lugar de las constantes numéricas, las constantes de carácter, etc., que representan dichas constantes simbólicas.

Se define una constante simbólica escribiendo

```
#define nombre texto
```

en donde *`nombre`* representa un nombre simbólico, que se suele escribir en letras mayúsculas, y *`texto`* representa la secuencia de caracteres asociada al nombre simbólico. Adviértase que *`texto`* no acaba con un punto y coma, ya que la definición de una constante simbólica no es una verdadera instrucción de C. Es más, si *`texto`* acabase con un punto y coma, este punto y coma se trataría como si fuese parte de la constante numérica, la constante de carácter o la constante de cadena de caracteres que se sustituye por el nombre simbólico.

EJEMPLO 2.31. Un programa en C contiene las siguientes definiciones de constantes simbólicas:

```
#define INTERES 0.23

#define PI 3.141593

#define TRUE 1
#define FALSE 0

#define AMIGA "Susana"
```

Nótese que los nombres simbólicos están escritos en mayúsculas, para distinguirlos de los identificadores ordinarios de C. Adviértase también que las definiciones no acaban en punto y coma.

Supóngase ahora que el programa contiene la instrucción

```
area = PI * radio * radio;
```

Durante el proceso de compilación, cada aparición de una constante simbólica será reemplazada por su correspondiente texto. Por tanto, la instrucción anterior se transformará en

```
area = 3.141593 * radio * radio;
```

Supongamos ahora que se ha incluido (incorrectamente) un punto y coma en la definición de `PI`, esto es,

```
#define PI 3.141593;
```

La instrucción de asignación de `area` se transformaría entonces en

```
area = 3.141593; * radio * radio;
```

Adviértase el punto y coma que precede al primer asterisco. Esto es claramente incorrecto, y provocará un error durante la compilación.

La sustitución de texto por una constante simbólica será llevada a cabo en cualquier sitio a continuación de la instrucción `#define`, *excepto* dentro de una cadena de caracteres. Por tanto, cualquier texto encerrado entre comillas (dobles) no se verá afectado por este proceso de sustitución.

EJEMPLO 2.32. Un programa en C contiene las siguientes instrucciones:

```
#define CONSTANTE 6.023E23
float c;
. . . . .
printf("CONSTANTE = %f", c);
```

La instrucción `printf` no se verá afectada por la definición de la constante simbólica, ya que el término `"CONSTANTE = %f"` es una constante de cadena de caracteres. Si, en cambio, se hubiera escrito la instrucción `printf` de la forma

```
printf("CONSTANTE = %f", CONSTANTE);
```

entonces la instrucción `printf` se habría transformado en

```
printf("CONSTANTE = %f", 6.023E23);
```

durante el proceso de compilación.

Las constantes simbólicas no son necesarias para escribir programas en C. Sin embargo, se recomienda su utilización, ya que contribuyen al desarrollo de programas claros y ordenados. Por ejemplo, las constantes simbólicas se identifican de forma más rápida que la información que representan, y los nombres simbólicos suelen sugerir el significado de sus datos asociados. Además, es mucho más fácil cambiar el valor de una única constante simbólica que cambiar toda aparición de alguna constante numérica que pueda aparecer en varios lugares del programa.

La existencia de `#define`, que se utiliza para definir constantes simbólicas, es una de las principales características incluidas en el *preprocesador* de C (un programa que se ocupa de un primer paso en la traducción de un programa en C a lenguaje máquina). El preprocesador de C se trata con mayor detalle en el Capítulo 14 (ver sección 14.6).

CUESTIONES DE REPASO

2.1. ¿Qué caracteres incluye el conjunto de caracteres de C?

2.2. Mencionar las reglas referentes a los nombres de los identificadores. ¿Son equivalentes las letras mayúsculas a las minúsculas? ¿Se pueden incluir dígitos en un identificador? ¿Se puede incluir cualquier carácter especial?

2.3. ¿Cuántos caracteres puede tener un identificador? ¿Son todos los caracteres significativos en igual forma?

2.4. ¿Qué es una palabra reservada en C? ¿Qué restricciones tiene su uso?

2.5. Mencionar y describir los cuatro tipos básicos de datos en C.

2.6. Mencionar y describir los cuatro cualificadores de tipos de datos. ¿A qué tipos de datos se puede aplicar cada cualificador?

2.7. Nombrar y describir los cuatro tipos básicos de constantes en C.

2.8. Mencionar todas las reglas que se aplican a todas las constantes de tipo numérico.

2.9. ¿Qué reglas especiales se les aplican a las constantes enteras?

2.10. Al escribir constantes enteras, ¿cómo se diferencian las constantes decimales, octales y hexadecimales?

2.11. ¿Cuál es, típicamente, el mayor valor permisible de una constante entera? Escribir la respuesta en decimal, octal y hexadecimal.

2.12. ¿Qué es una constante entera sin signo? ¿Y una constante entera larga? ¿En qué se diferencian de las constantes enteras ordinarias? ¿Cómo se pueden escribir y diferenciar?

2.13. Describir dos formas diferentes de escribir constantes en coma flotante. ¿Qué reglas especiales se aplican a cada caso?

2.14. ¿Cuál es el propósito del exponente (opcional) en una constante en coma flotante?

2.15. ¿Cuál es, típicamente, el mayor valor que puede tener una constante en coma flotante? Compárese con el de una constante entera.

2.16. ¿Cómo se pueden escribir e identificar las constantes en coma flotante de «simple precisión» y «largas»?

2.17. ¿Cuántas cifras significativas tiene como máximo, típicamente, una constante en coma flotante?

2.18. Describir las diferencias de precisión entre una constante entera y una en coma flotante. ¿En qué circunstancias se debe utilizar cada una de ellas?

2.19. ¿Qué es una constante de carácter? ¿En qué se diferencian las constantes de carácter de las constantes de tipo numérico? ¿Representan valores numéricos las constantes de carácter?

2.20. ¿Qué es el conjunto de caracteres ASCII? ¿Es su uso común?

2.21. ¿Qué es una secuencia de escape? ¿Qué propósito tiene?

2.22. Mencionar las secuencias de escape estándar en C. Describir otras secuencias de escape no estándar normalmente disponibles.

2.23. ¿Qué es una constante de cadena de caracteres? ¿En qué se diferencian las constantes de cadena de caracteres de las constantes de carácter? ¿Representan valores numéricos las constantes de cadena de caracteres?

2.24. ¿Se pueden incluir en una constante de cadena de caracteres secuencias de escape? Explicarlo.

2.25. ¿Qué es una variable? ¿Cómo se pueden caracterizar las variables?

2.26. ¿Qué es una variable array? ¿En qué se diferencia una variable array de una variable ordinaria?

2.27. ¿Qué restricción deben satisfacer todos los datos contenidos en un array?

2.28. ¿Cómo se pueden distinguir entre sí los elementos de un array?

2.29. ¿Qué es un índice? ¿Qué rango de valores puede tener el índice de un array unidimensional de *n* elementos?

2.30. ¿Cuál es el propósito de una declaración de tipo? ¿De qué consta una declaración de tipo?

2.31. ¿Se deben declarar todas las variables que aparecen en un programa en C?

2.32. ¿Cómo se les puede asignar valores iniciales a las variables en una declaración de tipo? ¿Cómo se les puede asignar cadenas de caracteres a arrays unidimensionales de tipo carácter?

2.33. ¿Qué es una expresión? ¿Qué clase de información es representada en una expresión?

2.34. ¿Qué es un operador? Describir varios tipos diferentes de operadores que estén incluidos en el lenguaje C.

2.35. Mencionar los tres tipos diferentes de instrucciones en C. Describir la composición de cada una de ellas.

2.36. ¿Se pueden incluir instrucciones dentro de otras? Explicarlo.

2.37. ¿Qué es una constante simbólica? ¿Cómo se define una constante simbólica? ¿Cómo se escribe la definición? ¿Dónde se debe poner la definición de una constante simbólica en un programa en C?

2.38. ¿Qué les ocurre a las constantes simbólicas que aparecen en un programa en C durante el proceso de compilación?

PROBLEMAS

2.39. Determinar cuáles de los siguientes son identificadores válidos. Si no son válidos, explicar por qué.

a) `registro1`
b) `1registro`
c) `archivo_3`
d) `return`
e) `$impuesto`
f) `nombre`
g) `nombre y direccion`
h) `nombre_y_direccion`
i) `nombre-y-direccion`
j) `123-45-6789`

2.40. Supongamos que la versión de C puede reconocer sólo los ocho primeros caracteres del nombre de un identificador, aunque los nombres de los identificadores puedan ser de longitud arbitraria. ¿Cuáles de los siguientes pares de nombres de identificadores se considerarán como idénticos y cuáles se distinguirán?

a) `nombre, nombres`
b) `direccion, Direccion`
c) `identificador_1, identificador_2`
d) `lista1, lista2`
e) `respuesta, RESPUESTA`
f) `car1, car_1`

2.41. Determinar cuáles de lo siguientes valores numéricos son constantes válidas. Si una constante es válida, especificar si es entera o real. Especificar también la base en que está escrita cada constante entera válida.

a) `0.5`
b) `27,822`
c) `9.3e12`
d) `9.3e-12`
e) `12345678`
f) `12345678L`
g) `0.8E+0.8`
h) `0.8E 8`
i) `0515`
j) `018CDF`
k) `0XBCFDAL`
l) `0x87e3ha`

2.42. Determinar cuáles de las siguientes son constantes de carácter válidas.

a) `'a'`
b) `'$'`
c) `'\n'`
d) `'/n'`
e) `'\\'`
f) `'\a'`
g) `'T'`
h) `'\0'`
i) `'xyz'`
j) `'\052'`

2.43. Determinar cuáles de las siguientes son constantes de cadena de caracteres válidas.

a) `'8:15 P.M.'`
b) `"Rojo, Blanco y Azul"`
c) `"Nombre:`
d) `"Capítulo 3 (Cont\'d)"`
e) `"1.3e-12"`
f) `"NEW YORK, NY 10020"`
g) `"El profesor dijo, "por favor no se duerman en clase"`

2.44. Escribir las declaraciones apropiadas para cada grupo de variables y arrays.

a) Variables enteras: `p, q`
Variables en coma flotante: `x, y, z`
Variables de carácter: `a, b, c`

b) Variables en coma flotante: `raiz1, raiz2`
Variable entera larga: `contador`
Variable entera corta: `indicador`

c) Variable entera: `indice`
Variable entera sin signo: `num_cliente`
Variables de doble precisión: `bruto, impuesto, neto`

d) Variables de carácter: `actual, ultimo`
Variable entera sin signo: `contador`
Variable en coma flotante: `error`

e) Variables de carácter: `primero, ultimo`
Array de caracteres de 80 elementos: `mensaje`

2.45. Escribir declaraciones apropiadas y asignar los valores iniciales dados para cada grupo de variables y arrays.

a) Variables en coma flotante: `a=-8.2, b=0.005`
Variables enteras: `x=129, y=87, z=-22`
Variables de carácter: `c1='w', c2='&'`

b) Variables de doble precisión: d1=2.88 x 10^{-8}, d2=-8.4 x 10^{5}
Variables enteras: `u=711` (octal), `v=ffff` (hexadecimal)

c) Variable entera larga: `grande=123456789`
Variable de doble precisión: `c=0.3333333333`
Variable de carácter: `eol=carácter de nueva línea`

d) Array unidimensional de caracteres: `mensaje="ERROR"`

2.46. Explicar el propósito de cada una de las siguientes expresiones.

a) `a - b`
b) `a * (b + c)`
c) `d = a * (b + c)`
d) `a >= b`
e) `(a % 5) == 0`
f) `a < (b / c)`
g) `--a`

2.47. Identificar cuándo cada una de las instrucciones siguientes es una instrucción de expresión, una instrucción compuesta o una instrucción de control

a)
```
a * (b + c)
```

b)
```
while (a < 100) {
     d = a * (b + c);
     ++a;

}
```

c)
```
if (x > 0)
     y = 2.0;
else
     y = 3.0;
```

d)
```
{
     ++x;
     if (x > 0)
          y = 2.0;
```

```
        else
            y = 3.0;
        printf("%f", y);
    }
```

e)
```
    {
        ++x;
        if (x > 0) {
            y = 2.0;
            z = 6.0;
        }
        else {
            y = 3.0;
            z = 9.0;
        }
    }
```

2.48. Escribir una definición apropiada para cada una de las siguientes constantes simbólicas, como si apareciesen en un programa en C.

	Constante	*Texto*
a)	FACTOR	-18
b)	ERROR	0.0001
c)	BEGIN	{
	END	}
d)	NOMBRE	"Adrián"
e)	EOLN	'\n'
f)	COSTE	"$19.95"

CAPÍTULO 3

Operadores y expresiones

Ya hemos visto que las constantes, variables, elementos de una formación y referencias a funciones se pueden unir con varios operadores para formar expresiones. También hemos mencionado que C posee un gran número de operadores que se pueden agrupar en diferentes categorías. En este capítulo examinaremos con detalle varias de esas categorías. Concretamente, veremos cómo utilizar operadores aritméticos, operadores unarios, operadores relacionales y lógicos, operadores de asignación y el operador condicional para formar expresiones.

Los datos sobre los que actúan los operadores se denominan *operandos*. Algunos operadores requieren dos operandos, mientras que otros actúan sólo sobre un operando. La mayoría de los operadores permiten que los operandos puedan ser expresiones. Existen algunos operadores que sólo permiten variables como operandos (lo veremos posteriormente).

3.1 . OPERADORES ARITMÉTICOS

Existen cinco *operadores aritméticos* en C:

Operador	Propósito
+	suma
-	resta
*	multiplicación
/	división
%	resto de división entera

El operador `%` es a veces denominado el operador *módulo*.

No hay operador de potenciación en C. Sin embargo, hay una *función de biblioteca* (`pow`) que realiza la potenciación (ver sección 3.6).

Los operandos sobre los que actúan los operadores aritméticos deben representar valores numéricos. Por tanto, los operandos deben ser cantidades enteras, en coma flotante o caracteres (recuérdese que las constantes de carácter representan valores enteros, los determinados por el conjunto de caracteres de la computadora). El operador de resto (`%`) requiere que los dos operandos sean enteros y el segundo operando no nulo. Análogamente, el operador de división (`/`) requiere que el segundo operando sea no nulo.

La división de una cantidad entera por otra es denominada *división entera*. Esta operación siempre tiene como resultado el cociente entero truncado (se desprecia la parte decimal del cocien-

te). Por otra parte, si una operación de división se lleva a cabo con dos números en coma flotante, o con un número en coma flotante y un entero, el resultado será un cociente en coma flotante.

EJEMPLO 3.1. Supóngase que unas variables `a` y `b` tienen valores `10` y `3`, respectivamente. Se muestran a continuación varias expresiones aritméticas en las que aparecen estas variables, acompañadas del resultado.

Expresión	*Valor*
`a + b`	`13`
`a - b`	`7`
`a * b`	`30`
`a / b`	`30`
`a % b`	`1`

Observe el cociente truncado resultante de la operación de división, ya que ambos operandos representan cantidades enteras. Observe también el resto entero resultante del uso del operador módulo en la última expresión.

Supongamos ahora que `v1` y `v2` son variables en coma flotante cuyos valores son `12.5` y `2.0`, respectivamente. Mostramos a continuación varias expresiones aritméticas en las que aparecen estas variables, acompañadas del resultado.

Expresión	*Valor*
`v1 + v2`	`14.5`
`v1 - v2`	`10.5`
`v1 * v2`	`25.0`
`v1 / v2`	`6,25`

Supongamos, finalmente, que `c1` y `c2` son variables de tipo carácter que representan los caracteres `P` y `T`, respectivamente. Se muestran a continuación varias expresiones aritméticas en las que aparecen estas variables, acompañadas de los resultados (basadas en el conjunto de caracteres ASCII).

Expresión	*Valor*
`c1`	`80`
`c1 + c2`	`164`
`c1 + c2 + 5`	`169`
`c1 + c2 + ´5´`	`217`

Observar que `P` está codificada como `80` (en decimal), `T` está codificada como `84` y `5` como `53` en el conjunto de caracteres ASCII, como se muestra en la Tabla 2.1.

Si uno o los dos operandos representan valores negativos, las operaciones de suma, resta, multiplicación y división tendrán como resultado valores cuyos signos están determinados por las reglas del álgebra. El resultado de la división estará truncado hacia cero, es decir, el resultado siempre será menor en valor absoluto que el verdadero cociente.

La interpretación de la operación de resto no está clara cuando uno de los operandos es

negativo. La mayoría de las versiones de C asignan al resto el mismo signo del primer operando. Por tanto, la condición

```
a = ((a /b) * b) + (a % b)
```

siempre se satisface, sin tener en cuenta los signos de los valores representados por `a` y `b`.

Los programadores sin experiencia deben tener cuidado con el uso de la operación de resto cuando uno de los operandos es negativo. En general es mejor evitar situaciones como ésta.

EJEMPLO 3.2. Supongamos que `a` y `b` son variables enteras cuyos valores son `11` y `-3`, respectivamente. A continuación se muestran varias expresiones aritméticas en las que aparecen estas variables, acompañadas de los resultados.

Expresión	*Valor*
`a + b`	`8`
`a - b`	`14`
`a * b`	`-33`
`a / b`	`-3`
`a % b`	`2`

Si se le ha asignado a `a` un valor de `-11` y a `b` `3`, entonces el valor de `a / b` aún sería `-3`, pero el valor de `a % b` sería `-2`. Análogamente, si `a` y `b` tienen asignados valores negativos (`-11` y `-3`, respectivamente), entonces el valor de `a / b` sería `3` y el valor de `a % b` sería `-2`.

Observe que la condición

```
a = ((a / b) * b) + (a % b)
```

se satisface en cada uno de los casos anteriores. La mayoría de las versiones de C determinarán el signo del resto de esta forma, aunque este punto está sin especificar en la definición formal del lenguaje.

EJEMPLO 3.3. A continuación se presentan los resultados obtenidos con operandos en coma flotante con diferentes signos. Supongamos que `r1` y `r2` son variables en coma flotante cuyos valores asignados son `-0.66` y `4.50`. A continuación se muestran varias expresiones aritméticas en las que aparecen estas variables, acompañadas de los resultados.

Expresión	*Valor*
`r1 + r2`	`3.84`
`r1 - r2`	`-5.16`
`r1 * r2`	`-2.97`
`r1 / r2`	`-0.146667`

Los operandos que difieren en el tipo pueden sufrir una conversión de tipo antes de que la expresión alcance su valor final. En general, el resultado final se expresará con la mayor precisión posible, de forma consistente con los tipos de datos de los operandos. Se pueden aplicar las reglas siguientes cuando ninguno de los operandos es `unsigned`.

1. Si los dos operandos son tipos en coma flotante con precisión distinta (por ejemplo un `float` y un `double`), el operando de menor precisión se transformará a la precisión del otro operando y el resultado se expresará con esta mayor precisión. Por tanto, una operación entre un `float` y un `double` dará como resultado un `double`; entre un `float` y un `long double` dará lugar a un `long double`; y entre un `long double` y un `double` se producirá un `long double`. (Nota: en algunas versiones de C, todos los operandos de tipo `float` se convierten automáticamente en `double`).
2. Si un operando es un tipo en coma flotante (por ejemplo `float`, `double` o `long double`) y el otro es un `char` o un `int` (incluyendo `short int` y `long int`), el `char` o `int` se convertirán al tipo en coma flotante del otro operando y el resultado se expresará de igual forma. Por tanto, una operación entre un `int` y un `double` tendrá como resultado un `double`.
3. Si ninguno de los operandos es del tipo en coma flotante pero uno es un `long int`, el otro se transformará en `long int` y el resultado será `long int`. Por consiguiente, una operación entre un `long int` y un `int` tendrá como resultado un `long int`.
4. Si ningún operando es del tipo en coma flotante ni `long int`, ambos operandos se convertirán en `int` (si es necesario) y el resultado será `int`. Así una operación entre un `short int` y un `int` tendrá como resultado un `int`.

Puede encontrar una relación más detallada de estas reglas en el Apéndice D. También aparecen las conversiones que involucran operandos `unsigned`.

EJEMPLO 3.4. Supongamos que `i` es una variable entera cuyo valor es `7`, `f` una variable en coma flotante cuyo valor es `5.5` y `c` una variable de tipo carácter que representa el carácter `w`. A continuación se muestran varias expresiones en las que aparecen estas variables. En cada expresión aparecen dos operandos de tipos diferentes. Se supone que se utiliza el conjunto de caracteres ASCII.

Expresión	*Valor*	*Tipo*
`i + f`	`12.5`	doble precisión
`i + c`	`126`	entero
`i + c - '0'`	`78`	entero
`(i + c) - (2 * f / 5)`	`123.8`	doble precisión

Observe que la `w` se codifica como `119` (en decimal) y el `0` como `48` en el conjunto de caracteres ASCII, como se muestra en la Tabla 2.1.

Si se desea, se puede convertir el valor resultante de una expresión a un tipo de datos diferente. Para hacer esto, la expresión debe ir precedida por el nombre del tipo de datos deseado, encerrado con paréntesis, esto es

`(`*`tipo de datos`*`)` *`expresión`*

Este tipo de construcción se denomina conversión de tipos («cast»).

EJEMPLO 3.5. Supongamos que i es una variable entera cuyo valor es 7 y f una variable en coma flotante con valor asignado 8.5. La expresión

```
(i + f) % 4
```

no es válida, porque el primer operando (i + f) es en coma flotante en vez de entero. Sin embargo, la expresión

```
((int) (i + f)) % 4
```

hace que el primer operando se transforme en entero y por tanto es válida, obteniéndose como resto de la división entera 3.

Observe que la especificación explícita de tipo se aplica sólo al primer operando, no a toda la expresión.

El tipo de datos asociado a la expresión en sí no es cambiado por un «cast». Es el *valor* de la expresión el que sufre la conversión de tipo cuando aparece el «cast». Esto tiene especial relevancia cuando la expresión consta de una sola variable.

EJEMPLO 3.6. Supongamos que f es una variable en coma flotante cuyo valor es 5.5. La expresión

```
((int) f) % 2
```

contiene dos operandos enteros y por tanto es válida, dando como resultado el resto entero 1. Sin embargo, observe que f sigue siendo una variable en coma flotante con un valor de 5.5, aunque el valor de f se convirtiese en un entero (5) al efectuar la operación del resto.

Los operadores de C se agrupan jerárquicamente de acuerdo con su *precedencia* (su orden de evaluación). Las operaciones con mayor precedencia se efectúan antes que las que tienen menor precedencia. Sin embargo, se puede alterar el orden natural de evaluación mediante el uso de paréntesis, como se muestra en el Ejemplo 3.5.

Entre los operadores aritméticos, *, / y % se encuentran dentro de un mismo grupo de precedencia, y + y - se encuentran en otro. El primer grupo tiene mayor precedencia que el segundo. Por tanto, las operaciones de multiplicación, división y resto se efectuarán antes que las de suma y resta.

Otra consideración importante a tener en cuenta es el *orden* en que se efectuarán operaciones consecutivas dentro del mismo grupo de precedencia. Esto se conoce como *asociatividad*. Dentro de cada uno de los grupos de precedencia descrito anteriormente, la asociatividad es de izquierda a derecha. En otras palabras, operaciones consecutivas de suma y resta se efectúan de izquierda a derecha, así como operaciones consecutivas de multiplicación, división y resto.

EJEMPLO 3.7. La expresión aritmética

```
a - b / c * d
```

es equivalente a la fórmula algebraica a-[(b/c)xd]. Por tanto, si las variables en coma flotante a, b, c y d tienen asignados los valores 1., 2., 3. y 4., respectivamente, la expresión tendría como valor -1.666666..., ya que

```
1.-[(2./3.) x 4.]=1.-[0.666666... x 4.]=1.-2.666666...=-1.666666...
```

Observe que se efectúa en primer lugar la división, ya que esta operación tiene una precedencia mayor que la resta. El cociente resultante se multiplica por `4.`, a causa de la asociatividad de izquierda a derecha. Entonces se resta el producto a `1`, obteniendo el valor final de `-1.666666...`

La precedencia natural de las operaciones se puede alterar mediante el uso de paréntesis, permitiendo éstos que se puedan efectuar operaciones aritméticas de una expresión en el orden que se desee. De hecho, se pueden *anidar* los paréntesis, es decir, un par dentro de otro. En estos casos se efectúan primero las operaciones más internas.

EJEMPLO 3.8. La expresión aritmética

```
(a - b) / (c * d)
```

es equivalente a la fórmula algebraica `(a-b)/(c x d)`. Por tanto, si las variables en coma flotante `a`, `b`, `c` y `d` tienen asignados los valores `1.`, `2.`, `3.` y `4.`, respectivamente, la expresión tendrá como valor `-0.08333333...`, ya que

```
(1.-2.)/(3. x 4.)=-1./12.=-0.08333333...
```

Compárese este resultado con el obtenido en el Ejemplo 3.7.

A veces es una buena idea utilizar paréntesis para hacer más clara una expresión, aunque no sean necesarios. Por otra parte, se deben evitar en lo posible expresiones sobrecargadas como las del siguiente ejemplo. Expresiones como ésta son difíciles de leer, y a menudo se escriben de forma incorrecta por los paréntesis no emparejados.

EJEMPLO 3.9. Consideremos la expresión aritmética

```
2 * ((i % 5) * (4 + (j - 3) / (k + 2)))
```

donde `i`, `j` y `k` son variables enteras. Si se les asigna a estas variables los valores `8`, `15` y `4`, respectivamente, la expresión anterior se evaluaría como sigue:

```
2 x ((8%5) x (4+(15-3)/(4+2)))=2 x (3 x (4+(12/6)))=
2 x (3 x (4+2))= 2 x (3 x 6)=2 x 18=36
```

Supongamos que se le asigna el valor de esta expresión a la variable entera w, esto es,

```
w = 2 * ((i % 5) * (4 + (j - 3) / (k + 2)));
```

Es mejor, por norma general, fragmentar esta larga expresión aritmética en varias expresiones más cortas, como

```
u = i % 5;
v = 4 + (j - 3) / (k + 2);
w = 2 * (u * v);
```

donde u y v son variables enteras. Es más difícil cometer errores en la escritura de estas expresiones equivalentes que en la larga expresión original.

Las expresiones de asignación se tratarán con detalle en la sección 3.4.

3.2. OPERADORES UNARIOS

C incluye una clase de operadores que actúan sobre un solo operando para producir un nuevo valor. Estos operadores se denominan *operadores unarios* o *monarios*. Los operadores unarios suelen preceder a su único operando, aunque algunos operadores unarios se escriben detrás de su operando.

Es probable que el operador unario de uso más frecuente sea el *menos unario*, que consiste en un signo menos delante de una constante numérica, una variable o una expresión. (Algunos lenguajes de programación permiten que se incluya el signo menos como parte de una constante numérica. Sin embargo, en C todas las constantes numéricas son positivas. Por tanto, un número negativo es en realidad una expresión, que consiste en el operador unario menos, seguido de una constante numérica positiva.)

Adviértase que la operación menos unaria es distinta del operador aritmético que representa la resta (–). El operador resta requiere dos operandos.

EJEMPLO 3.10. He aquí varios ejemplos que ilustran el uso de la operación menos unaria.

```
-743        -0X7FFF     -0.2              -5E-8
-raiz1      -(x + y)    -3 * (x + y)
```

En cada caso, el signo menos es seguido por un operando numérico que puede ser una constante entera, una constante en coma flotante, una variable numérica o una expresión aritmética.

Otros dos operadores unarios de uso frecuente son el *operador incremento*, ++, y el *operador decremento*, --. El operador incremento hace que su operando se incremente en uno, mientras que el operador decremento hace que su operando se decremente en uno. El operando utilizado con cada uno de estos operadores debe ser una variable simple.

EJEMPLO 3.11. Supongamos que i es una variable entera que tiene asignado el valor 5. La expresión ++i, que es equivalente a escribir i = i + 1, hace que el valor de i sea 6. Análogamente la expresión --i, que es equivalente a i = i - 1, hace que el valor (partiendo del original) de i pase a ser 4.

Los operadores incremento y decremento se pueden utilizar, cada uno de ellos, de dos formas distintas, dependiendo de si el operador se escribe delante o detrás del operando. Si el operador *precede* al operando (por ejemplo ++i), el valor del operando se modificará *antes* de que se utilice con otro propósito. Sin embargo, si el operador *sigue* al operando (por ejemplo i++), entonces el valor del operando se modificará *después* de ser utilizado.

EJEMPLO 3.12. Un programa en C incluye una variable entera i, cuyo valor inicial es 1. Supongamos que el programa incluye las tres siguientes instrucciones printf. (Para una breve explicación de la instrucción printf, véase el Ejemplo 1.6.)

```
printf("i = %d\n", i);
printf("i = %d\n", ++i);
printf("i = %d\n", i);
```

Estas instrucciones `printf` generarán las tres líneas siguientes. (Cada instrucción `printf` genera una línea.)

```
i = 1
i = 2
i = 2
```

La primera instrucción hace que se visualice el valor original de `i`. La segunda instrucción incrementa `i` y presenta después su valor. La última instrucción visualiza el valor final de `i`.

Supongamos ahora que el programa incluye las tres siguientes instrucciones `printf`, en lugar de las tres dadas anteriormente.

```
printf("i = %d\n", i);
printf("i = %d\n", i++);
printf("i = %d\n", i);
```

Observe que la primera y tercera instrucción son idénticas a las mostradas anteriormente. Sin embargo, en la segunda instrucción el operador unario sigue a la variable entera en lugar de precederla.

Estas instrucciones generarán las tres líneas siguientes.

```
i = 1
i = 1
i = 2
```

La primera instrucción hace que se visualice el valor original de `i` como en el caso anterior. La segunda instrucción hace que se visualice el valor actual de `i (1)` y después lo incrementa (a 2). La última instrucción visualiza el valor final de `i (2)`.

Hablaremos mucho más del uso de la instrucción `printf` en el Capítulo 4. Por ahora, fijémonos simplemente en la diferencia entre la expresión `++i` en el primer grupo de instrucciones y la expresión `i++` en el segundo grupo.

Otro operador unario que merece ser citado ahora es el operador `sizeof`. Este operador devuelve el tamaño de su operando en bytes. El operador `sizeof` siempre precede a su operando. El operando puede ser una expresión o puede ser un «cast».

En programas sencillos no se suele utilizar este operador. Sin embargo, este operador permite determinar el número de bytes asignados a diferentes tipos de datos. Esta información puede ser muy útil cuando se transfiere el programa a una computadora diferente o a una nueva versión de C. También se utiliza para la asignación dinámica de la memoria, como se explica en la sección 10.4.

EJEMPLO 3.13. Supongamos que `i` es una variable entera, `x` una variable en coma flotante, `d` una variable de doble precisión y `c` una variable de tipo carácter. Las instrucciones

```
printf("Entero: %d\n", sizeof i);
printf("Coma flotante: %d\n", sizeof x);
```

```
printf("Doble precisión: %d\n", sizeof d);
printf("Carácter: %d\n", sizeof c);
```

generarían la siguiente salida:

```
Entero: 2
Coma flotante: 4
Doble precisión: 8
Carácter: 1
```

Vemos por tanto que esta versión de C reserva 2 bytes para cada cantidad entera, 4 bytes para cada cantidad en coma flotante, 8 bytes para cada cantidad de doble precisión y 1 byte para cada carácter. Estos valores pueden variar de una versión de C a otra, como se explica en la sección 2.3.

Otra forma de conseguir la misma información es utilizar un «cast» en lugar de una variable dentro de cada instrucción `printf`. De acuerdo con esto, podríamos escribir las instrucciones `printf` de la forma siguiente:

```
printf("Entero: %d\n", sizeof (integer));
printf("Coma flotante: %d\n", sizeof (float));
printf("Doble precisión: %d\n", sizeof (double));
printf("Carácter: %d\n", sizeof (char));
```

Estas instrucciones `printf` generarán una salida igual a las anteriores. Nótese que cada «cast» está encerrado entre paréntesis, como se vio en la sección 3.1.

Consideremos finalmente la declaración

```
char texto[] = "California";
```

La instrucción

```
printf("Número de caracteres = %d", sizeof texto);
```

generará la siguiente salida.

```
Número de caracteres = 11
```

Por tanto vemos que la formación `texto` contiene 11 caracteres, como se explicó en el Ejemplo 2.26.

Un «cast» se puede considerar también como un operador unario (ver Ejemplo 3.5 y la anterior discusión). En términos generales, una referencia al operador «cast» se escribe así: (*tipo*). Por tanto, los operadores unarios que hemos visto hasta el momento en este libro son `-`, `++`, `--`, `sizeof` y (*tipo*).

Los operadores unarios tienen mayor precedencia que los operadores aritméticos. Por tanto, si un operador unario menos actúa sobre una expresión aritmética que contiene uno o más operadores aritméticos, la operación unaria menos se efectuará primero (a menos, por supuesto, que la expresión aritmética esté entre paréntesis). La asociatividad de los operadores unarios es también de izquierda a derecha, aunque es raro que aparezcan en programas sencillos operadores unarios consecutivos.

EJEMPLO 3.14. Supongamos que x e y son variables enteras con valores asignados de 10 y 20, respectivamente. El valor de la expresión -x+y será -10+20=-10. Observe que la operación unaria menos se efectúa antes que la suma.

Supongamos ahora que introducimos paréntesis, de forma tal que tenemos la expresión -(10+20). El valor de esta expresión es -(10+20)=-30. Notar que ahora la suma *precede* a la operación unaria menos.

C incluye otros operadores unarios. Se irán tratando en secciones posteriores de este libro según se vayan haciendo necesarios.

3.3. OPERADORES RELACIONALES Y LÓGICOS

En C existen cuatro *operadores relacionales*:

Operador	*Significado*
<	menor que
<=	menor o igual que
>	mayor que
>=	mayor o igual que

Estos operadores se encuentran dentro del mismo grupo de precedencia, que es menor que la de los operadores unarios y aritméticos. La asociatividad de estos operadores es de izquierda a derecha.

Muy asociados a los operadores relacionales, existen dos *operadores de igualdad*:

Operador	*Significado*
==	igual que
!=	no igual que

Los operadores de igualdad se encuentran en otro grupo de precedencia, por debajo de los operadores relacionales. La asociatividad de estos operadores es también de izquierda a derecha.

Estos seis operadores se utilizan para formar expresiones lógicas que representan condiciones que pueden ser verdaderas o falsas. La expresión resultante será de tipo entero, ya que *verdadero* es representado por el valor entero 1 y *falso* por el valor 0.

EJEMPLO 3.15. Supongamos que i, j y k son variables enteras con valores asignados 1, 2 y 3, respectivamente. A continuación se muestran varias expresiones lógicas en las que aparecen estas variables.

Expresión	*Interpretación*	*Valor*
i < j	verdadero	1
(i + j) >= k	verdadero	1
(j + k) > (i + 5)	falso	0
k != 3	falso	0
j == 2	verdadero	1

Cuando se efectúan operaciones relacionales y de igualdad, si los operandos son de diferente tipo, se convertirán de acuerdo con las reglas citadas en la sección 3.1.

EJEMPLO 3.16. Supongamos que `i` es una variable entera cuyo valor es `7`, `f` es una variable en coma flotante cuyo valor es `5.5` y `c` es una variable de carácter que representa el carácter `'w'`. A continuación se muestran varias expresiones lógicas que hacen uso de estas variables. En cada expresión aparecen dos tipos diferentes de operandos. (Se supone que se utiliza el conjunto de caracteres ASCII.)

Expresión	*Interpretación*	*Valor*
`f > 5`	verdadero	1
`(i + f) <= 10`	falso	0
`c == 119`	verdadero	1
`c != 'p'`	verdadero	1
`c >= 10 * (i + f)`	falso	0

Además de los operadores relacionales y de igualdad, C posee dos *operadores lógicos* (denominados también *conectivas lógicas*). Estos son:

Operador	*Significado*
`&&`	y
`||`	o

Estos operadores se denominan *y lógica* y *o lógica*, respectivamente.

Los operadores lógicos actúan sobre operandos que son a su vez expresiones lógicas. Permiten combinar expresiones lógicas individuales, formando otras condiciones lógicas más complicadas que pueden ser verdaderas o falsas. El resultado de una operación *y lógica* será verdadero sólo si los dos operandos son verdaderos, mientras que el resultado de una operación *o lógica* será verdadero si alguno de los dos operandos es verdadero o ambos a la vez. En otras palabras, el resultado de una operación *o lógica* será falso sólo si los dos operandos son falsos.

En este contexto, *cualquier* valor no nulo, no sólo el `1`, se interpretará como verdadero.

EJEMPLO 3.17. Supongamos que `i` es una variable entera cuyo valor es `7`, `f` una variable en coma flotante cuyo valor es `5.5` y `c` una variable de carácter que representa el carácter `'w'`. A continuación se muestran varias expresiones lógicas complejas en las que aparecen estas variables.

Expresión	*Interpretación*	*Valor*
`(i >= 6) && (c == 'w')`	verdadero	1
`(i >= 6) || (c == 119)`	verdadero	1
`(f < 11) && (i > 100)`	falso	0
`(c != 'p') || ((i + f) <= 10)`	verdadero	1

La primera expresión es verdadera porque los dos operandos son verdaderos. En la segunda expresión, los dos operandos también son verdaderos, por tanto toda la expresión es verdadera. La tercera expresión es falsa porque el segundo operando es falso. Y, finalmente, la cuarta expresión es verdadera porque el primer operando es verdadero.

Cada uno de los operadores lógicos pertenece a su propio grupo de precedencia. La *y lógica* tiene mayor precedencia que la *o lógica*. Los dos grupos de precedencia se encuentran por debajo del grupo que contiene los operadores de igualdad. La asociatividad es de izquierda a derecha. Más adelante se relacionan los grupos de precedencia.

C también incluye el operador unario `!`, que niega el valor de una expresión lógica; es decir, hace que una expresión que era originalmente verdadera se haga falsa y viceversa. Este operador se denomina operador de *negación lógica* (o *no lógico*).

EJEMPLO 3.18. Supongamos que `i` es una variable entera con valor `7` y `f` es una variable en coma flotante con valor `5.5`. A continuación se muestran varias expresiones lógicas en las que aparecen estas variables y el operador de negación lógica.

Expresión	*Interpretación*	*Valor*
`f > 5`	verdadero	1
`!(f > 5)`	falso	0
`i <= 3`	falso	0
`!(i <= 3)`	verdadero	1
`i > (f + 1)`	verdadero	1
`!(i > (f + 1))`	falso	0

En próximos capítulos de este libro veremos otros ejemplos en los que aparece el operador de negación lógica.

La jerarquía de precedencia de operadores que abarca todos los operadores discutidos hasta ahora se va haciendo cada vez más extensa. A continuación se muestra una relación de las precedencias de los operadores, de mayor a menor.

Categoría de operador	*Operadores*	*Asociatividad*
operadores unarios	`- ++ -- ! sizeof (tipo)`	D → I
multiplicación, división y resto aritméticos	`* / %`	I → D
suma y resta aritméticas	`+ -`	I → D
operadores relacionales	`< <= > >=`	I → D
operadores de igualdad	`== !=`	I → D
y lógica	`&&`	I → D
o lógica	`||`	I → D

Más adelante se muestra una lista más completa en la Tabla 3.1.

EJEMPLO 3.19. Consideremos de nuevo las variables `i`, `f` y `c`, como se han descrito en los Ejemplos 3.16 y 3.17; es decir, `i=7`, `f=5.5` y `c='w'`. A continuación se muestran varias expresiones lógicas que hacen uso de estas variables.

Expresión	*Interpretación*	*Valor*
`i + f <= 10`	falso	0
`i >= 6 && c == 'w'`	verdadero	1
`c != 'p' || i + f <= 10`	verdadero	1

Cada una de estas expresiones se ha presentado ya antes (la primera en el Ejemplo 3.16 y las otras dos en el Ejemplo 3.17), aunque se habían incluido paréntesis en los ejemplos anteriores. Los paréntesis no son necesarios a causa de las precedencias propias de los operadores. Por consiguiente, las operaciones aritméticas se efectuarán automáticamente antes que las operaciones relacionales o de igualdad, y las operaciones relacionales y de igualdad se efectuarán automáticamente antes que las conectivas lógicas.

Consideremos la última expresión en particular. La primera operación que se efectuará será la suma `(i + f)`; después la comparación relacional `(i + f <= 10)`; después la comparación de igualdad `(c != 'p')`, y finalmente la condición *o lógica*.

Las expresiones lógicas compuestas que constan de expresiones lógicas individuales unidas por los operadores lógicos `&&` y `||` se evalúan de izquierda a derecha, pero sólo hasta que se ha establecido el valor verdadero/falso del conjunto. Por tanto, una expresión lógica compuesta no se evaluará completamente si su valor se puede establecer a partir de la evaluación de algunos de sus operandos.

EJEMPLO 3.20. Consideremos la expresión lógica compuesta que se muestra a continuación.

```
error > .0001 && cont < 100
```

Si `error > .0001` es falso, entonces el segundo operando `(cont < 100)` no se evaluará, ya que la expresión entera tendrá que ser necesariamente falsa.

Por otra parte, supongamos que hemos escrito la expresión

```
error > .0001 || cont < 100
```

Si `error > .0001` es verdadero, entonces la expresión completa será verdadera. Por tanto, no se evaluará el segundo operando. Sin embargo, si `error > .0001` es falso entonces, la segunda expresión `(cont < 100)` debe ser evaluada para determinar si toda la expresión es verdadera o falsa.

3.4. OPERADORES DE ASIGNACIÓN

Existen varios operadores de asignación en C. Todos se utilizan para formar expresiones de asignación, en las que se asigna el valor de una expresión a un identificador.

El operador de asignación más usado es =. Las expresiones de asignación que utilizan este operador se escriben de la siguiente forma:

`identificador = expresión`

donde *`identificador`* representa generalmente una variable y *`expresión`* una constante, una variable o una expresión más compleja.

EJEMPLO 3.21. He aquí algunas expresiones de asignación que hacen uso del operador =.

```
a = 3
x = y
delta = 0.001
suma = a + b
area = longitud * anchura
```

La primera expresión de asignación hace que se le asigne a la variable `a` el valor `3` y la segunda asignación hace que se le asigne el valor de `y` a `x`. En la tercera asignación, el valor en coma flotante `0.001` se le asigna a `delta`. En las dos últimas asignaciones se le asigna a una variable el resultado de una expresión (el valor de `a + b` se le asigna a `suma` y el valor de `longitud * anchura` a `area`).

Recuerde que el *operador de asignación* = y el *operador de igualdad* == son distintos. El operador de asignación se utiliza para asignar un valor a un identificador, mientras que el operador de igualdad se usa para determinar si dos expresiones tienen el mismo valor. No se pueden utilizar estos operadores de forma indistinta. Es frecuente que algunos programadores, cuando están aprendiendo, utilicen de forma incorrecta el operador de asignación cuando quieren comprobar una igualdad. El resultado de esto es un error lógico que suele ser difícil de detectar.

Las expresiones de asignación se suelen llamar *instrucciones de asignación*, ya que se suelen escribir como instrucciones completas. Sin embargo, también se pueden escribir expresiones de asignación como expresiones que están incluidas dentro de otras instrucciones (veremos más sobre esto en próximos capítulos).

Si los dos operandos de una expresión de asignación son de tipo de datos diferentes, el valor de la expresión de la derecha (el operando de la derecha) se convertirá automáticamente al tipo del identificador de la izquierda. De esta forma, toda la expresión de asignación será del mismo tipo de datos.

En determinados casos, esta conversión automática de tipo puede conllevar a una alteración del dato que se está asignando. Por ejemplo:

- Un valor en coma flotante puede ser truncado si se asigna a un identificador entero.
- Un valor de doble precisión puede ser redondeado si se asigna a un identificador en coma flotante (de simple precisión).
- Una cantidad entera puede ser alterada si es asignada a un identificador de un entero más corto o a un identificador de carácter (se pueden perder algunos de los bits más significativos).

Además, cuando se asigne a un identificador de tipo numérico el valor de una constante de carácter, este valor dependerá del conjunto de caracteres que se esté utilizando. De esto pueden resultar inconsistencias entre distintas versiones de C.

El uso descuidado de conversiones de tipo es una fuente de errores frecuente entre los programadores noveles.

EJEMPLO 3.22. En las siguientes expresiones de asignación, supongamos que `i` es una variable de tipo entero.

Expresión	Valor
`i = 3.3`	3
`i = 3.9`	3
`i = -3.9`	–3

Supongamos ahora que `i` y `j` son variables de tipo entero y que a `j` se le ha asignado el valor `5`. Mostramos a continuación varias expresiones que hacen uso de estas dos variables.

Expresión	*Valor*	
`i = j`	5	
`i = j / 2`	2	
`i = 2 * j / 2`	5	(asociatividad de izquierda a derecha)
`i = 2 * (j / 2)`	4	(división truncada, seguida de multiplicación)

Finalmente, supongamos que `i` es una variable de tipo entero y que estamos utilizando el conjunto de caracteres ASCII.

Expresión	*Valor*
`i = 'x'`	120
`i = '0'`	48
`i = ('x' - '0') / 3`	24
`i = ('y' - '0') / 3`	24

En C están permitidas asignaciones múltiples de la forma

identificador 1 = *identificador 2* = ... = *expresión*

En estos casos, las asignaciones se efectúan de derecha a izquierda. Por tanto, la asignación múltiple

identificador 1 = *identificador 2* = *expresión*

es equivalente a

identificador 1 = (*identificador 2* = *expresión*)

y así sucesivamente con anidaciones de derecha a izquierda para asignaciones múltiples.

EJEMPLO 3.23. Supongamos que `i` y `j` son variables enteras. La expresión de asignación múltiple

```
i = j = 5
```

hará que a `i` y `j` se les asigne el valor entero `5`. (Para ser más exactos, primero se le asigna `5` a `j` y después se le asigna el valor de `j` a `i`.)

Análogamente, la expresión de asignación múltiple

```
i = j = 5.9
```

hará que a `i` y `j` se les asigne el valor entero `5`. Recordar que cuando se le asigna a la variable entera `j` el valor en coma flotante `5.9`, éste es truncado.

C posee, además, los cinco siguientes operadores de asignación: `+=`, `-=`, `*=`, `/=` y `%=`. Para ver cómo se utilizan, consideremos el primer operador, `+=`. La expresión de asignación

expresión 1 += *expresión 2*

es equivalente a

```
expresión 1 = expresión 1 + expresión 2
```

De forma análoga, la expresión de asignación

```
expresión 1 -= expresión 2
```

es equivalente a

```
expresión 1 = expresión 1 - expresión 2
```

y de igual forma para los cinco operadores.

Normalmente, `expresión 1` es un identificador tal como una variable o un elemento de una formación.

EJEMPLO 3.24. Supongamos que `i` y `j` son variables enteras con valores asignados de `5` y `7`, y `f` y `g` variables en coma flotante con valores `5.5` y `-3.25`. A continuación se muestran varias expresiones de asignación que hacen uso de estas variables. Cada expresión utiliza los valores *originales* de `i`, `j`, `f` y `g`.

Expresión	*Expresión equivalente*	*Valor final*
`i += 5`	`i = i + 5`	`10`
`f -= g`	`f = f - g`	`8.75`
`j *= (i - 3)`	`j = j * (i - 3)`	`14`
`f /= 3`	`f = f / 3`	`1.833333`
`i %= (j - 2)`	`i = i % (j - 2)`	`0`

Los operadores de asignación tienen menor precedencia que el resto de los operadores que hemos discutido anteriormente. Por tanto, las operaciones unarias, aritméticas, relacionales, de igualdad y lógicas se efectúan antes que las operaciones de asignación. Además, las operaciones de asignación tienen asociatividad de derecha a izquierda.

La jerarquía de operadores, atendiendo a su precedencia, que presentamos en la última sección, se puede modificar como sigue para incluir los operadores de asignación.

Categoría de operador	*Operadores*	*Asociatividad*
operadores unarios	`- ++ -- ! sizeof (tipo)`	D → I
multiplicación, división y resto aritméticos	`* / %`	I → D
suma y resta aritméticas	`+ -`	I → D
operadores relacionales	`< <= > >=`	I → D
operadores de igualdad	`== !=`	I → D
y lógica	`&&`	I → D
o lógica	`||`	I → D
operadores de asignación	`= += -= *= /= %=`	D → I

La Tabla 3.1 contiene una lista completa.

EJEMPLO 3.25. Supongamos que x, y y z son variables enteras que tienen asignados los valores 2, 3 y 4, respectivamente. La expresión

```
x *= -2 * (y + z) / 3
```

es equivalente a la expresión

```
x = x * (-2 * (y + z) / 3)
```

Ambas expresiones harán que se le asigne a x el valor -8.

Consideremos el orden en el que se efectúan las operaciones en la primera expresión. Las operaciones aritméticas preceden a la instrucción de asignación. Por tanto se evaluará primero la expresión (y + z) con un resultado de 7. Después el valor de esta expresión se multiplicará por -2, obteniéndose -14. Este producto se dividirá a continuación por 3 y será truncado, con lo que resulta -4. Finalmente este cociente truncado se multiplica por el valor original de x (2), con lo que se obtiene el resultado final -8.

Observe que todas las operaciones aritméticas explícitas se efectúan antes de que se realicen la multiplicación final y la asignación.

C posee otros operadores de asignación además de los que hemos descrito. Los trataremos en el Capítulo 13.

3.5. EL OPERADOR CONDICIONAL

Se pueden efectuar operaciones condicionales simples con el *operador condicional* (? :). Una expresión que hace uso del operador condicional se denomina *expresión condicional.* Se puede escribir una instrucción de este tipo en lugar de la instrucción más tradicional if-else, que se tratará en el Capítulo 6.

Una expresión condicional se escribe de la forma siguiente:

expresión 1 ? expresión 2 : expresión 3

Cuando se evalúa una expresión condicional, *expresión 1* es evaluada primero. Si *expresión 1* es verdadera (si su valor es no nulo), entonces *expresión 2* es evaluada y éste es el valor de la expresión condicional. Sin embargo, si *expresión 1* es falsa (si su valor es cero), entonces se evalúa *expresión 3* y éste es el valor de la expresión condicional. Nótese que sólo se evalúa una de las expresiones (*expresión 2* o *expresión 3*) cuando se determina el valor de una expresión condicional.

EJEMPLO 3.26. En la expresión condicional que se muestra a continuación, supongamos que i es una variable entera.

```
(i < 0) ? 0 : 100
```

Se evalúa primero la expresión (i < 0). Si es verdadera (si el valor de i es menor que 0), el valor de toda la expresión condicional es 0. En cualquier otro caso (si el valor de i no es menor que 0), el valor de toda la expresión condicional es 100.

Supongamos que `f` y `g` son variables en coma flotante en la siguiente expresión condicional.

```
(f < g) ? f : g
```

Esta expresión condicional toma el valor de `f` si `f` es menor que `g`; de otra forma, la expresión condicional toma el valor de `g`. En otras palabras, la expresión condicional devuelve el valor de la menor de las dos variables.

Si los operandos (*`expresión 2`* y *`expresión 3`*) son de tipos diferentes, el tipo de datos de la expresión condicional se determinará de acuerdo con las reglas dadas en la sección 3.1.

EJEMPLO 3.27. Supongamos ahora que `i` es una variable entera y que `f` y `g` son variables en coma flotante. En la expresión condicional

```
(f < g) ? i : f
```

aparecen operandos de tipo entero y en coma flotante. El tipo de dato de la expresión condicional será en coma flotante, aun cuando se seleccione el valor de `i` como valor de la expresión (por la regla 2 de la sección 3.1).

Las expresiones condicionales suelen aparecer en la parte derecha de una instrucción de asignación simple. Se le asigna al identificador de la izquierda el valor resultante de la expresión condicional.

EJEMPLO 3.28. La siguiente es una instrucción de asignación que contiene una expresión condicional en la parte de la derecha.

```
indicador = (i < 0) ? 0 : 100
```

Si el valor de `i` es negativo, se le asignará a `indicador` el valor `0`. Si `i` no es negativo, se le asignará a `indicador` el valor `100`.

La siguiente es otra instrucción de asignación que contiene una expresión condicional en la parte derecha.

```
min = (f < g) ? f : g
```

Esta instrucción hace que se le asigne a `min` el menor valor de `f` y `g`.

El operador condicional tiene su propia precedencia, justamente superior a los operadores de asignación. La asociatividad es de derecha a izquierda.

La Tabla 3.1 resume las precedencias de todos los operadores tratados en este capítulo.

En el Apéndice C se presenta una lista completa de todos los operadores de C, la cual es más amplia que la mostrada en la Tabla 3.1.

EJEMPLO 3.29. En la siguiente instrucción de asignación, `a`, `b` y `c` se suponen variables enteras. En la instrucción aparecen operadores de seis grupos de precedencia distintos.

```
c += (a > 0 && a <= 10) ? ++a : a/b;
```

La instrucción comienza por la evaluación de la expresión compuesta

```
(a > 0 && a <= 10)
```

Tabla 3.1. Grupos de precedencia de operadores

Categoría de operador	Operadores	Asociatividad
operadores unarios	- ++ -- ! sizeof (tipo)	D → I
multiplicación, división y resto aritméticos	* / %	I → D
suma y resta aritméticas	+ -	I → D
operadores relacionales	< <= > >=	I → D
operadores de igualdad	== !=	I → D
y lógica	&&	I → D
o lógica	||	I → D
operador condicional	?:	D → I
operadores de asignación	= += -= *= /= %=	D → I

Si esta expresión es cierta, se evalúa la expresión `++a`. Si no es así, se evalúa la expresión `a/b`. Finalmente se efectúa la operación de asignación `(+=)`, haciendo que se incremente `c` en el valor de la expresión condicional.

Si, por ejemplo, `a`, `b` y `c` tienen los valores `1`, `2` y `3`, respectivamente, entonces el valor de la expresión condicional será `2` (porque será evaluada la expresión `++a`) y a `c` se le asignará el valor `5 (c = 3 + 2)`. Por otro lado, si los valores de `a`, `b` y `c` fuesen `50`, `10` y `20`, respectivamente, entonces el valor de la expresión condicional sería `5` (porque se evaluaría la expresión `a/b`) y el valor de `c` pasaría a ser `25 (c = 20 + 5)`.

3.6. FUNCIONES DE BIBLIOTECA

El lenguaje C se acompaña de un cierto número de *funciones de biblioteca* que realizan varias operaciones y cálculos de uso frecuente. Estas funciones de biblioteca no son parte del lenguaje en sí, pero las incluyen todas las implementaciones del lenguaje. Algunas funciones devuelven un dato en su llamada; otras indican cuándo una determinada condición es verdadera o falsa, devolviendo un valor de `1` o `0`, respectivamente; y otras efectúan operaciones específicas sobre los datos y no devuelven nada. Suelen existir funciones de biblioteca para efectuar las operaciones que son dependientes de la computadora.

Por ejemplo, hay funciones de biblioteca que efectúan las operaciones estándar de entrada/salida (leer y escribir caracteres, leer y escribir números, abrir y cerrar archivos, comprobar la condición de fin de archivo, etc.), funciones que realizan operaciones sobre caracteres (convertir minúsculas en mayúsculas, determinar si un carácter es una letra mayúscula, etc.), funciones que realizan operaciones en cadenas de caracteres (copiar una cadena de caracteres en otra, comparar dos cadenas, concatenar dos cadenas, etc.), y funciones que realizan diversos cálculos matemáticos (evaluación de funciones trigonométricas, logarítmica y exponencial, cálculo de valores absolutos, raíces cuadradas, etc.). También existen funciones de biblioteca de otros tipos.

Las funciones de biblioteca de propósitos relacionados se suelen encontrar agrupadas en programas objeto en archivos de biblioteca separados. Estos archivos de biblioteca se proporcionan como parte de cada compilador de C. Todos los compiladores de C contienen los mismos grupos de funciones de biblioteca, aunque no existe una normalización precisa. Por tanto,

puede existir alguna diferencia en las funciones de biblioteca disponibles en diferentes versiones del lenguaje.

Un conjunto de funciones de biblioteca típico incluirá un gran número de funciones comunes para la mayoría de los compiladores de C, tales como las mostradas en la Tabla 3.2. En esta tabla, la columna «tipo» se refiere al tipo de datos del resultado que devuelve la función. El tipo *void* en la función `srand` indica que la función no devuelve nada.

Tabla 3.2. Algunas funciones de biblioteca de uso común

Función	Tipo	Propósito
`abs(i)`	`int`	Devolver el valor absoluto de `i`.
`ceil(d)`	`double`	Redondear por exceso al entero más próximo (el entero más pequeño que sea mayor o igual a `d`).
`cos(d)`	`double`	Devolver el coseno de `d`.
`cosh(d)`	`double`	Devolver el coseno hiperbólico de `d`.
`exp(d)`	`double`	elevar *e* a la potencia `d` (`e=2.7182818...` es la base del sistema logarítmico natural (Neperiano)).
`fabs(d)`	`double`	Devolver el valor absoluto de `d`.
`floor(d)`	`double`	Redondear por defecto al entero más próximo (el entero más grande que no sea mayor que `d`).
`fmod(d1,d2)`	`double`	Devolver el resto de `d1/d2` (parte no entera del cociente), con el mismo signo que `d1`.
`getchar()`	`int`	Introducir un carácter desde el dispositivo de entrada estándar.
`log(d)`	`double`	Devolver el logaritmo natural de `d`.
`pow(d1,d2)`	`double`	Devolver `d1` elevado a la potencia `d2`.
`printf(...)`	`int`	Mandar datos al dispositivo de salida estándar (los argumentos son complicados; ver Capítulo 4).
`putchar(c)`	`int`	Mandar un carácter al dispositivo de salida estándar.
`rand()`	`int`	Devolver un entero positivo aleatorio.
`sin(d)`	`double`	Devolver el seno de `d`.
`sqrt(d)`	`double`	devolver la raíz cuadrada de `d`.
`srand(u)`	`void`	Inicializar el generador de números aleatorios.
`scanf(...)`	`int`	Introducir datos del dispositivo de entrada estándar (los argumentos son complicados; ver Capítulo 4).
`tan(d)`	`double`	Devolver la tangente de `d`.
`toascii(c)`	`int`	Convertir el valor del argumento a ASCII.
`tolower(c)`	`int`	Convertir una letra a minúscula.
`toupper(c)`	`int`	Convertir una letra a mayúscula.

Nota: Tipo se refiere al tipo de datos del resultado devuelto por la función.

`c` indica argumento de tipo carácter.
`i` indica argumento de tipo entero.
`d` indica argumento de doble precisión.
`u` indica argumento entero sin signo.

En el Apéndice H se incluye una lista más larga, que incorpora todas las funciones de biblioteca que se utilizan en los programas ejemplo presentados en este libro. Para conseguir una lista completa, el lector debe recurrir al manual de referencia del programador de su versión de C.

Se accede a una función de biblioteca escribiendo simplemente el nombre de la función, seguido de una lista de *argumentos* que representan información que se le pasa a la función. Los argumentos se deben encontrar encerrados entre paréntesis y separados por comas. Pueden ser constantes, nombres de variables o expresiones más complicadas. Los paréntesis deben estar presentes, aunque no haya argumentos.

Una función que devuelve un dato puede aparecer en cualquier sitio dentro de una expresión, en lugar de una constante o un identificador (una variable o un elemento de una formación). Se puede acceder a una función que efectúa operaciones sobre datos pero que no devuelve ningún valor simplemente escribiendo el nombre de la función, ya que este tipo de referencia a una función constituye una instrucción de expresión.

EJEMPLO 3.30. Se muestra a continuación un fragmento de un programa en C que calcula las raíces de la ecuación cuadrática

$$ax^2 + bx + c = 0$$

utilizando la fórmula

$$x = \frac{-b \pm (b^2 - 4ac)^{1/2}}{2a}$$

Este programa utiliza la función de biblioteca `sqrt` para evaluar la raíz cuadrada.

```
main()        /* solución de una ecuación cuadrática */
{
    double a, b, c, raiz, x1, x2;

    /* leer valores de a, b y c */

    raiz = sqrt(b * b - 4 * a * c);
    x1 = (-b + raiz) / (2 * a);
    x2 = (-b - raiz) / (2 * a);

    /* escribir valores de a, b, c, x1 y x2 */
}
```

Para utilizar una función de biblioteca puede ser necesario incluir cierta información dentro de la parte principal del programa. Por ejemplo, las declaraciones de funciones y definiciones de constantes simbólicas suelen necesitarse cuando se utilizan funciones de biblioteca (ver secciones 7.3, 8.5 y 8.6). Esta información suele encontrarse almacenada en ciertos archivos que se proporcionan con el compilador. Por tanto, la información requerida se puede obtener simplemente accediendo a estos archivos. Esto se lleva a cabo mediante la instrucción del preprocesador `#include`, que es

```
#include <nombre_archivo>
```

en donde *`nombre_archivo`* representa el nombre de un determinado archivo.

Los nombres de estos archivos especiales son específicos de cada implementación de C, aunque hay ciertos nombres de archivos comúnmente usados, como `stdio.h` y `math.h`. El sufijo «h» generalmente designa un archivo de «cabecera», que indica que se debe incluir al comienzo del programa. (En la sección 8.6 se discuten los archivos de cabecera.)

Nótese la similitud entre la instrucción del preprocesador `#include` y la instrucción del preprocesador `#define`, que se discutió en la sección 2.9.

EJEMPLO 3.31. Conversión de un carácter de minúscula a mayúscula. El siguiente es un programa completo en C que lee una letra minúscula, la transforma en mayúscula y la escribe.

```
/* leer una minúscula y escribir la mayúscula correspondiente */

#include <stdio.h>
#include <ctype.h>

main()
{
    int minusc, mayusc;

    minusc = getchar();
    mayusc = toupper(minusc);
    putchar(mayusc);
}
```

En este programa aparecen tres funciones de biblioteca: `getchar`, `toupper` y `putchar`. Las dos primeras devuelven un solo carácter (`getchar` devuelve un carácter que se introduce por el teclado y `toupper` devuelve el carácter de la mayúscula correspondiente a su argumento). La última función (`putchar`) hace que se visualice el valor de su argumento. Nótese que las dos últimas funciones tienen un argumento, mientras que la primera no tiene ninguno, como se indica con los paréntesis vacíos.

Nótese también las instrucciones del preprocesador `#include <stdio.h>` y `#include <ctype.h>`, que aparecen al comienzo del programa. Estas instrucciones hacen que se inserten los contenidos de los archivos `stdio.h` y `ctype.h` en el programa al comienzo del proceso de compilación. La información contenida en dichos archivos es esencial para el funcionamiento correcto de las funciones de biblioteca `getchar`, `putchar` y `toupper`.

CUESTIONES DE REPASO

3.1. ¿Qué es una expresión? ¿Cuáles son sus componentes?

3.2. ¿Qué es un operador? Describir varios tipos diferentes de operadores de C.

3.3. ¿Qué es un operando? ¿Cuál es la relación entre operadores y operandos?

3.4. Describir los cinco operadores aritméticos de C. Mencionar las reglas asociadas a su utilización.

3.5. Mencionar las reglas que se aplican a expresiones con operandos de tipos distintos.

3.6. ¿Cómo se puede cambiar el valor de una expresión a un tipo de datos diferente? ¿Cómo se llama a esto?

3.7. ¿Qué se entiende por precedencia de operadores? ¿Cuáles son las precedencias relativas de los operadores aritméticos?

3.8. ¿Qué se entiende por asociatividad? ¿Cuál es la asociatividad de los operadores aritméticos?

3.9. ¿Cuándo se deben incluir paréntesis en una expresión? ¿Cuándo se debe evitar el uso de paréntesis?

3.10. ¿En qué orden se efectúan las operaciones en una expresión que contiene paréntesis anidados?

3.11. ¿Qué es un operador unario? ¿Cuántos operandos van asociados a un operador unario?

3.12. Describir los seis operadores unarios tratados en este capítulo. ¿Cuál es el propósito de cada uno de ellos?

3.13. Describir dos formas distintas de utilizar los operadores de incremento y decremento. ¿Cuál es la diferencia entre ellas?

3.14. ¿Cómo es la precedencia de los operadores unarios en relación con la de los operadores aritméticos? ¿Cuál es su asociatividad?

3.15. ¿Cómo se puede determinar el número de bytes que ocupa cada tipo de datos en un determinado compilador de C?

3.16. Describir los cuatro operadores relacionales de C. ¿Con qué tipo de operandos se pueden utilizar? ¿Qué tipo de expresión se obtiene?

3.17. Describir los dos operadores de igualdad de C. ¿En qué se diferencian de los operadores relacionales?

3.18. Describir los dos operadores lógicos de C. ¿Cuál es el propósito de cada uno de ellos? ¿Con qué tipo de operandos se pueden utilizar? ¿Qué tipo de expresión se obtiene?

3.19. ¿Cuáles son las precedencias relativas de los operadores relacionales, de igualdad y lógicos entre sí y respecto a los operadores aritméticos y unarios? ¿Cuáles son sus asociatividades?

3.20. Describir el operador *no lógico* (negación lógica). ¿Cuál es su propósito? ¿En qué grupo de precedencia está incluido? ¿Cuántos operandos requiere? ¿Cuál es su asociatividad?

3.21. Describir los seis operadores de asignación tratados en este capítulo. ¿Cuál es el propósito de cada uno de ellos?

3.22. ¿Cómo se determina el tipo de una expresión de asignación cuando los dos operandos son de tipos diferentes? ¿En qué sentido puede ser esto a veces una fuente de errores de programación?

3.23. ¿Cómo se pueden escribir múltiples asignaciones en C? ¿En qué orden se efectuarán las asignaciones?

3.24. ¿Cuál es la precedencia de los operadores de asignación en relación con otros operadores? ¿Cuál es su asociatividad?

3.25. Describir el uso del operador condicional para formar expresiones condicionales. ¿Cómo se evalúa una expresión condicional?

3.26. ¿Cómo se determina el tipo de una expresión condicional cuando sus operandos son de tipos diferentes?

3.27. ¿Cómo se puede combinar el operador condicional con el operador de asignación para formar una instrucción del tipo «if-else»?

3.28. ¿Cuál es la precedencia del operador condicional en relación con los otros operadores descritos en este capítulo? ¿Cuál es su asociatividad?

3.29. Describir, en términos generales, las clases de operaciones y cálculos realizados por las funciones de biblioteca de C.

3.30. ¿Forman realmente parte del lenguaje C las funciones de biblioteca? Explicarlo.

3.31. ¿Cómo se suelen encontrar agrupadas generalmente las funciones de biblioteca en un compilador de C?

3.32. ¿Cómo se accede a las funciones de biblioteca? ¿Cómo se pasa información a una función de biblioteca desde el punto de acceso?

3.33. ¿Qué es un argumento? ¿Cómo se escriben los argumentos? ¿Cómo se escribe una llamada a una función de biblioteca si no tiene argumentos?

3.34. ¿Cómo se almacena la información que pueden requerir las funciones de biblioteca? ¿Cómo se introduce esta información en un programa en C?

3.35. ¿Dentro de qué categoría general se encuentran las instrucciones `#define` y `#include`?

PROBLEMAS

3.36. Supongamos que `a`, `b` y `c` son variables enteras que tienen asignados los valores `a=8`, `b=3` y `c=-5`. Determinar el valor de cada una de las siguientes expresiones aritméticas.

a) `a + b + c`
b) `2 * b + 3 * (a -c)`
c) `a / b`
d) `a % b`
e) `a / c`
f) `a % c`
g) `a * b / c`
h) `a * (b / c)`
i) `(a * c) % b`
j) `a * (c % b)`

3.37. Supongamos que `x`, `y` y `z` son variables en coma flotante que tienen asignados los valores `x=88`, `y=3.5` y `z=-5.2`. Determinar el valor de cada una de las siguientes expresiones aritméticas.

a) `x + y + z`
b) `2 * y + 3 * (x - z)`
c) `x / y`
d) `x % y`
e) `x / (y + z)`
f) `(x / y) + z`
g) `2 * x / 3 * y`
h) `2 * x / (3 * y)`

3.38. Supongamos `c1`, `c2` y `c3` variables de tipo carácter que tienen asignados los caracteres `E`, `5` y `?`, respectivamente. Determinar el valor numérico de las siguientes expresiones, basándose en el conjunto de caracteres ASCII (ver Tabla 2.1).

a) `c1`
b) `c1 - c2 + c3`
c) `c2 - 2`
d) `c2 - '2'`
e) `c3 + '#'`
f) `c1 % c3`
g) `'2' + '2'`
h) `(c1 / c2) * c3`
i) `3 * c2`
j) `'3' * c2`

3.39. Un programa en C contiene las siguientes declaraciones:

```
int i, j;
long ix;
short s;
float x;
double dx;
char c;
```

Determinar el tipo de datos de cada una de las siguientes expresiones:

a) `i + c`
b) `x + c`
c) `dx + x`
d) `((int) dx) + ix`
e) `i + x`
f) `s + j`
g) `ix + j`
h) `s + c`
i) `ix + c`

3.40. Un programa en C contiene las siguientes declaraciones y asignaciones iniciales:

```
int i = 8, j = 5;
float x = 0.005, y = -0.01;
char c = 'c', d = 'd';
```

Determinar el valor de cada una de las siguientes expresiones. Utilizar los valores asignados inicialmente a las variables para cada expresión.

a) `(3 * i - 2 * j) % (2 * d - c)`
b) `2 * ((i / 5) + (4 * (j - 3)) % (i + j - 2))`
c) `(i - 3 * j) % (c + 2 * d) / (x - y)`
d) `-(i + j)`
e) `++i`
f) `i++`
g) `--j`
h) `++x`
i) `y--`
j) `i <= j`
k) `c > d`
l) `x >= 0`
m) `x < y`
n) `j != 6`
o) `c == 99`
p) `5 * (i + j) > 'c'`
q) `(2 * x + y) == 0`
r) `2 * x + (y == 0)`
s) `2 * x + y == 0`
t) `!(i <= j)`
u) `!(c == 99)`
v) `!(x > 0)`

w) `(i > 0) && (j < 5)`
x) `(i > 0) || (j < 5)`
y) `(x > y) && (i > 0) || (j < 5)`
z) `(x > y) && (i > 0) || (j < 5)`

3.41. Un programa en C contiene las siguientes declaraciones y asignaciones iniciales:

```
int i = 8, j = 5, k;
float x = 0.005, y = -0.01, z;
char a, b, c = 'c', d = 'd';
```

Determinar el valor de cada una de las siguientes expresiones de asignación. Utilizar para cada expresión el valor inicial asignado a las variables.

a) `k = (i + j)`
b) `z = (x + y)`
c) `i = j`
d) `k = (x + y)`
e) `k = c`
f) `z = i / j`
g) `a = b = d`
h) `i = j = 1.1`
i) `z = k = x`
j) `k = z = x`
k) `i += 2`
l) `y -= x`
m) `x *= 2`
n) `i /= j`
o) `i %= j`
p) `i += (j -2)`
q) `k = (j == 5) ? i : j`
r) `k = (j > 5) ? i : j`
s) `z = (x >= 0) ? x : 0`
t) `z = (y >= 0) ? y : 0`
u) `a = (c < d) ? c : d`
v) `i -= (j > 0) ? j : 0`

3.42. Cada una de las siguientes expresiones utiliza una función de biblioteca. Identificar el propósito de cada expresión. (Ver en el Apéndice H una lista de las funciones de biblioteca.)

a) `abs(i - 2 * j)`
b) `fabs(x + y)`
c) `isprint(c)`
d) `isdigit(c)`
e) `toupper(d)`
f) `ceil(x)`
g) `floor(x + y)`
h) `islower(c)`
i) `isupper(j)`
j) `exp(x)`
k) `log(x)`
l) `sqrt(x*x + y*y)`
m) `isalnum(10 * j)`
n) `isalpha(10 * j)`
o) `isascii(10 * j)`
p) `toascii(10 * j)`
q) `fmod(x, y)`
r) `tolower(65)`
s) `pow(x - y, 3.0)`
t) `sin(x - y)`
u) `strlen("hola\0")`
v) `strpos("hola\0", 'e')`

3.43. Un programa en C contiene las siguientes declaraciones y asignaciones iniciales:

```
int i = 8, j = 5;
double x = 0.005, y = -0.01;
char c = 'c', d = 'd';
```

Determinar el valor de cada una de las siguientes expresiones, que hacen uso de funciones de biblioteca. (Ver en el Apéndice C una lista de las funciones de biblioteca.)

a) `abs(i - 2 * j)`
b) `fabs(x + y)`
c) `isprint(c)`
d) `isdigit(c)`
e) `toupper(d)`
f) `ceil(x)`
g) `ceil(x + y)`
h) `floor(x)`
i) `floor(x + y)`
j) `islower(c)`
k) `isupper(j)`
l) `exp(x)`
m) `log(x)`
n) `log(exp(x))`
o) `sqrt(x*x + y*y)`
p) `isalnum(10 * j)`
q) `isalpha(10 * j)`
r) `isascii(10 * j)`
s) `toascii(10 * j)`
t) `fmod(x, y)`
u) `tolower(65)`
v) `pow(x - y, 3.0)`
w) `sin(x - y)`
x) `strlen("hola\0")`
y) `strpos("hola\0", 'e')`
z) `sqrt(sin(x) + cos(y))`

3.44. Determinar de cuáles de las funciones de biblioteca mostradas en el Apéndice H dispone su compilador de C. ¿Dispone de algunas de las funciones con un nombre distinto? ¿Qué archivos de cabecera se requieren?

CAPÍTULO 4

Entrada y salida de datos

Ya hemos visto que el lenguaje C va acompañado de una colección de funciones de biblioteca que incluye un cierto número de funciones de entrada/salida. En este capítulo utilizaremos seis de estas funciones: `getchar`, `putchar`, `scanf`, `printf`, `gets` y `puts`. Estas seis funciones permiten la transferencia de información entre la computadora y los dispositivos de entrada/salida estándar (por ejemplo, un teclado y un monitor). Las dos primeras funciones, `getchar` y `putchar`, permiten la transferencia de caracteres individuales hacia dentro y hacia fuera de la computadora; `scanf` y `printf` son más complicadas, pero permiten la transferencia de caracteres individuales, valores numéricos y cadenas de caracteres; `gets` y `puts` permiten la entrada y salida de cadenas de caracteres. Una vez que hayamos aprendido el uso de estas funciones, seremos capaces de escribir un cierto número de programas en C sencillos pero completos.

4.1. INTRODUCCIÓN

Se puede acceder a una función de entrada/salida desde cualquier sitio de un programa con simplemente escribir el nombre de la función, seguido de una lista de argumentos entre paréntesis. Los argumentos representan los datos que le son enviados a la función. Algunas funciones de entrada/salida no requieren argumentos, pero deben aparecer los paréntesis vacíos.

Los nombres de estas funciones que devuelven datos pueden aparecer dentro de expresiones, como si cada referencia a una función fuese una variable ordinaria (por ejemplo `c = getchar();`), o se pueden referenciar como instrucciones separadas (por ejemplo `scanf(...);`). Algunas funciones no devuelven ningún dato. Estas funciones se referencian como si fuesen instrucciones separadas (por ejemplo `putchar();`).

La mayoría de las versiones de C incluyen una colección de archivos de cabecera que proporcionan la información necesaria (por ejemplo constantes simbólicas) para las distintas funciones de biblioteca. Cada archivo contiene generalmente la información necesaria para la utilización de un determinado grupo de funciones de biblioteca. Estos archivos se incluyen en un programa mediante la instrucción `#include` al comienzo del programa. Como norma general, el archivo de cabecera requerido para la entrada/salida estándar se llama `stdio.h` (consulte la sección 8.6 para más información sobre el contenido de estos archivos de cabecera).

EJEMPLO 4.1. El siguiente es un esquema de un programa de C típico que hace uso de varias rutinas de entrada/salida de la biblioteca estándar de C.

```
/* ejemplo del uso de funciones de biblioteca de entrada/salida */

#include <stdio.h>

main()
{
   char c, d;                        /* declaraciones */
   float x, y;
   int i, j, k;

   c = getchar();                    /* entrada de un carácter */
   scanf("%f", &x);                  /* entrada de número en coma
                                        flotante */
   scanf("%d %d", &i, &j);           /* entrada de enteros */
   . . .                             /* instrucciones de acción */
   putchar(d);                       /* salida de un carácter */
   printf("%3d %7.4f", k, y);        /* salida de números */
}
```

El programa comienza con la instrucción del preprocesador `#include <stdio.h>`. Esta instrucción hace que se incluya en el programa el contenido del archivo de cabecera `stdio.h`. Este archivo proporciona la información referente a las funciones de biblioteca `scanf` y `printf`. (La sintaxis de la instrucción `#include` puede variar de una versión de C a otra; algunas versiones del lenguaje utilizan comillas en lugar de los paréntesis en ángulo, por ejemplo `#include "stdio.h"`.)

A continuación de la instrucción del preprocesador se encuentra el encabezamiento del programa `main()` y algunas declaraciones de variables. Aparecen varias instrucciones de entrada/salida en las líneas que siguen a las declaraciones. En concreto, la instrucción de asignación `c = getchar();` hace que se lea un solo carácter del teclado y se le asigne a la variable `c`. La primera referencia a `scanf` hace que se lea por teclado un valor en coma flotante y se le asigne a la variable `x`, mientras que la segunda llamada a `scanf` hace que se lean del teclado dos cantidades enteras decimales y se les asignen a las variables `i` y `j`, respectivamente.

Las instrucciones de salida se comportan de forma análoga. La llamada a `putchar` hace que se visualice el valor de la variable de carácter `d`. Análogamente, la referencia a `printf` hace que se visualicen el valor de la variable entera `k` y el de la variable en coma flotante `y`.

Los detalles de cada instrucción de entrada/salida se discutirán en las siguientes secciones de este capítulo. Por ahora es suficiente conseguir una visión general de las instrucciones de entrada/salida incluidas en el programa anterior.

4.2. ENTRADA DE UN CARÁCTER - LA FUNCIÓN `getchar`

Mediante la función de biblioteca `getchar` se puede conseguir la entrada de caracteres uno a uno. Ya hemos visto en los Capítulos 1 y 2 y en el Ejemplo 4.1 algún caso en el que se utilizaba esta función. Ahora la veremos con más detalle.

La función `getchar` es parte de la biblioteca de C de entrada/salida estándar. Devuelve un carácter leído del dispositivo de entrada estándar (típicamente un teclado). La función no requiere argumentos, aunque es necesario que un par de paréntesis vacíos sigan a la palabra `getchar`.

En forma general, una referencia a la función `getchar` se escribe así:

```
variable de carácter = getchar();
```

donde *variable de carácter* es alguna variable de carácter previamente declarada.

EJEMPLO 4.2. Un programa en C contiene las siguientes instrucciones.

```
char c;
. . . . .
c = getchar();
```

En la primera instrucción se declara la variable `c` de tipo carácter. La segunda instrucción hace que se lea del dispositivo de entrada estándar un carácter y entonces se le asigne a `c`.

Si se encuentra una condición de *fin de archivo* («*end of file*») cuando se está leyendo un carácter con la función `getchar`, la función devolverá de forma automática el valor de la constante simbólica `EOF`. (Este valor se define dentro del archivo `stdio.h`. Normalmente `EOF` tendrá asignado el valor `-1`, aunque puede variar de un compilador a otro.) La detección de `EOF` de esta forma hace posible descubrir el fin de archivo en el momento y lugar que ocurra. Se puede entonces actuar en consecuencia. Ambas, la detección de la condición `EOF` y la acción oportuna, se pueden efectuar utilizando la instrucción `if-else` que se describe en el Capítulo 6.

La función `getchar` también se puede utilizar para leer cadenas de varios caracteres, leyendo en un bucle la cadena carácter a carácter. En el Ejemplo 4.4 veremos cómo se puede hacer esto. Hay más ejemplos en capítulos posteriores de este libro.

4.3. SALIDA DE UN CARÁCTER - LA FUNCIÓN `putchar`

Se puede visualizar un carácter utilizando la función de biblioteca `putchar`. Esta función es complementaria a la de entrada de un carácter `getchar`, que hemos visto en la sección anterior. También hemos visto ejemplos de utilización de estas dos funciones en los Capítulos 1 y 2 y en el Ejemplo 4.1. Ahora veremos `putchar` con más detalle.

La función `putchar`, así como `getchar`, es parte de la biblioteca de entrada/salida estándar. Transmite un carácter al dispositivo de salida estándar (típicamente un monitor). El carácter que se transmite estará representado normalmente por una variable de tipo carácter. Se debe proporcionar como argumento de la función, encerrado entre paréntesis, siguiendo a la palabra `putchar`.

En general, una referencia a la función `putchar` se escribe como sigue:

```
putchar(variable de carácter)
```

donde *`variable de carácter`* hace referencia a una variable de tipo carácter previamente declarada.

EJEMPLO 4.3. Un programa en C contiene las siguientes instrucciones.

```
char c;
. . . . .
putchar(c);
```

En la primera instrucción se declara la variable c de tipo carácter. La segunda instrucción hace que se transmita el valor actual de c al dispositivo de salida estándar (por ejemplo un monitor) en donde se visualizará. (Compárese con el Ejemplo 4.2, que ilustra el uso de la función getchar.)

La función `putchar` se puede utilizar para visualizar una constante de cadena de caracteres almacenando la cadena dentro de un array unidimensional de tipo carácter, como se explicó en el Capítulo 2. Se pueden escribir entonces mediante un bucle los caracteres uno a uno. La forma más cómoda de hacer esto es utilizando una instrucción `for`, como se puede ver en el siguiente ejemplo. (La instrucción `for` se trata con detalle en el Capítulo 6.)

EJEMPLO 4.4. Conversión de un texto de minúsculas a mayúsculas. El siguiente es un programa completo que lee una línea de texto en minúsculas, la almacena en un array de tipo carácter unidimensional y después la escribe en mayúsculas.

```
/* leer una línea un texto en minúsculas y escribirla en mayúsculas */

#include <stdio.h>
#include <ctype.h>

main()
{
   char letras[80];
   int cont, auxiliar;

   /* leer la línea */

   for (cont = 0; (letras[cont] = getchar()) != '\n'; ++cont)
       ;

   /* guardar el contador de caracteres */

   auxiliar = cont;

   /* escribir la línea en mayúsculas */

   for (cont = 0; cont < auxiliar; ++cont)
       putchar(toupper(letras[cont]));
}
```

Observe la declaración

```
char letras[80];
```

Aquí se declara `letras` como un array de tipo carácter de 80 elementos, los cuales representarán los caracteres de la línea de texto.

Consideremos ahora la instrucción

```
for (cont = 0; (letras[cont] = getchar()) != '\n'; ++cont)
    ;
```

Esta instrucción crea un bucle que hace que se introduzcan los caracteres en la computadora y los asigna a los elementos del array. El bucle comienza asignando a `cont` el valor cero. Se lee un carácter del dispositivo de entrada estándar y se le asigna a `letras[0]` (el primer elemento de `letras`). El valor de `cont` se incrementa en uno y entonces se repite el proceso para el siguiente elemento del array. El bucle continúa ejecutándose hasta que se introduzca el carácter de *nueva línea* (`'\n'`). El carácter de *nueva línea* indicará el final del texto y finalizará por tanto el proceso.

Una vez que se han introducido todos los caracteres, se le asigna a `auxiliar` el valor de `cont` correspondiente al último carácter. Comienza después otro bucle `for`, en el que se visualizan en el dispositivo de salida estándar las letras mayúsculas correspondientes a los caracteres introducidos. Los caracteres que eran originalmente dígitos, mayúsculas, signos de puntuación, etc., se visualizarán en su forma original. Por tanto, si introducimos

```
¡Es hora de que todos los hombres de bien acudan en ayuda de su pais!
```

la salida correspondiente será

```
¡ES HORA DE QUE TODOS LOS HOMBRES DE BIEN ACUDAN EN AYUDA DE SU PAIS!
```

Nótese que `auxiliar` tendrá asignado el valor 69, ya que tras el signo de fin de exclamación se encuentra el carácter de nueva línea.

El Capítulo 6 contiene información más detallada del uso de la instrucción `for` con arrays de caracteres. Por ahora, sólo es necesario entender de forma general qué es lo que hace el programa.

4.4. INTRODUCCIÓN DE DATOS - LA FUNCIÓN `scanf`

Se pueden introducir datos en la computadora procedentes del dispositivo de entrada estándar mediante la función de la biblioteca de C `scanf`. Esta función se puede utilizar para introducir cualquier combinación de valores numéricos, caracteres individuales y cadenas de caracteres. La función devuelve el número de datos que se han conseguido introducir correctamente.

En términos generales, la función `scanf` se escribe

```
scanf(cadena de control, arg1, arg2,..., argn)
```

donde *`cadena de control`* hace referencia a una cadena de caracteres que contiene cierta información sobre el formato de los datos y *`arg1`*, *`arg2`*,..., *`argn`* son argumentos que representan los datos. (En realidad, los argumentos representan *punteros* que indican las *direcciones* de memoria en donde se encuentran los datos. En el Capítulo 10 se trata esto con más detalle.)

En la cadena de control se incluyen grupos individuales de caracteres, con un grupo de caracteres por cada dato de entrada. Cada grupo de caracteres debe comenzar con el signo de porcentaje (`%`). En su forma más sencilla, un grupo de caracteres estará formado por el signo de porcentaje, seguido de un *carácter de conversión* que indica el tipo de dato correspondiente.

Dentro de la cadena de control, los diferentes grupos de caracteres pueden estar seguidos o pueden estar separados por caracteres de espaciado (espacios en blanco, tabuladores o caracteres de nueva línea). Si se utilizan los caracteres de espaciado para separar grupos de caracteres en la cadena de control, entonces todos los caracteres de espaciado consecutivos en los datos de entrada se leerán, pero serán ignorados. Es muy frecuente el uso de espacios en blanco como separadores de grupos de caracteres.

La Tabla 4.1 contiene los caracteres de conversión de uso más frecuente.

Tabla 4.1. Caracteres de conversión de los datos de entrada de uso común

Carácter de conversión	Significado
`c`	el dato es un carácter
`d`	el dato es un entero decimal
`e`	el dato es un valor en coma flotante
`f`	el dato es un valor en coma flotante
`g`	el dato es un valor en coma flotante
`h`	el dato es un entero corto
`i`	el dato es un entero decimal, octal o hexadecimal
`o`	el dato es un entero octal
`s`	el dato es una cadena de caracteres seguida de un carácter de espaciado (se añade automáticamente el carácter nulo `\0` al final)
`u`	el dato es un entero decimal sin signo
`x`	el dato es un entero hexadecimal
`[...]`	el dato es una cadena de caracteres que puede incluir caracteres de espaciado (ver explicación a continuación)

Los argumentos pueden ser variables o arrays, y sus tipos deben coincidir con los indicados por los grupos de caracteres correspondientes en la cadena de control. *Cada nombre de variable debe ser precedido por un ampersand* (`&`). (En realidad los argumentos son punteros que indican dónde se encuentran situados los datos en la memoria de la computadora, como se explica en el Capítulo 10.) Sin embargo, los nombres de arrays *no* deben ir precedidos por el ampersand.

EJEMPLO 4.5. La siguiente es una aplicación típica de la función `scanf`.

```
#include <stdio.h>

main()
```

```
{
   char concepto[20];
   int num_partida;
   float coste;

   . . . . .

   scanf("%s %d %f", concepto, &num_partida, &coste);

   . . . . .

}
```

Dentro de la función `scanf`, la cadena de control es `"%s %d %f"`. Contiene tres grupos de caracteres. El primer grupo de caracteres, `%s`, indica que el primer argumento (`concepto`) representa a una cadena de caracteres. El segundo grupo de caracteres, `%d`, indica que el segundo argumento (`&num_partida`) representa un valor entero decimal, y el tercer grupo de caracteres, `%f`, indica que el tercer argumento (`&coste`) representa un valor en coma flotante.

Observe que las variable numéricas `num_partida` y `coste` van precedidas por ampersands dentro de la función `scanf`. Sin embargo, delante de `concepto` no hay ampersand, ya que `concepto` es el nombre de un array.

Observe también que se podría haber escrito la función `scanf`

```
scanf("%s%d%f", concepto, &no_partida, &coste);
```

sin caracteres de espaciado en la cadena de control. Esto también es válido, aunque los datos de entrada se podrían haber interpretado de forma diferente al utilizar conversiones tipo `c` (más sobre esto más adelante en este capítulo).

Los datos pueden ser valores numéricos, caracteres individuales, cadenas de caracteres o alguna combinación de éstos. Se introducen del dispositivo de entrada estándar (típicamente un teclado). Los datos deben corresponderse con los argumentos de la función `scanf` en número, en tipo y en orden. Los datos numéricos deben estar escritos de la misma forma que las constantes numéricas (ver sección 2.4), aunque los valores octales no necesitan ir precedidos por un `0`, y los valores hexadecimales tampoco necesitan ir precedidos por `0x` o `0X`. Los valores en coma flotante deben incluir o un punto decimal o un exponente (o ambos).

Si se introducen dos o más datos, deben estar separados por caracteres de espaciado. (Una posible excepción a esta regla ocurre con las conversiones tipo `c`, como se describe en la sección 4.5) Los datos pueden encontrarse en dos o más líneas, ya que el carácter de nueva línea se considera como un carácter de espaciado y puede por tanto separar datos consecutivos.

Es más, si la cadena de control empieza por la lectura de un dato tipo carácter, es generalmente una buena idea preceder el primer carácter de conversión con un espacio en blanco. Esto hace que la función `scanf` ignore los caracteres extraños que se pueden haber introducido con anterioridad (por ejemplo, al pulsar la tecla `Intro` tras una línea de datos anterior).

EJEMPLO 4.6. Consideremos una vez más el esquema del programa en C mostrado en el Ejemplo 4.5, esto es,

```
#include <stdio.h>

main()
```

```
{
    char concepto[20];
    int num_partida;
    float coste;

    . . . . .

    scanf(" %s %d %f", concepto, &num_partida, &coste);

    . . . . .
}
```

Observe el espacio en blanco que precede `%s`. Esto evita la asignación a `concepto` de caracteres extraños introducidos con anterioridad.

Se podrían haber introducido los siguientes datos por el dispositivo de entrada estándar cuando se ejecutase el programa.

```
cremallera 12345 0.05
```

Se le asignarían a los diez primeros elementos del array `concepto` los caracteres que integran la cadena `cremallera`; a `num_partida` se le asignaría el valor entero `12345`, y a `coste` le sería asignado el valor en coma flotante `0.05`.

Observe que los datos se introducen en una línea, separados por espacios en blanco. También se podrían haber introducido los datos en líneas separadas, ya que los caracteres de nueva línea se considerarían también como caracteres de espaciado. Por tanto, los datos de entrada se podrían haber escrito así:

```
cremallera          cremallera          cremallera     12345
12345               12345    0.05       0.05
0.05
```

Observe que la conversión de caracteres tipo `s` se aplica a una cadena de caracteres que acaba en un carácter de espaciado. Por tanto, una cadena de caracteres que *incluye* caracteres de espaciado no se puede introducir de esta forma. Por supuesto, existen formas de trabajar que incluyen caracteres de espaciado. Una forma es utilizar la función `getchar` dentro de un bucle, como se vio en el Ejemplo 4.4. También se puede utilizar la función `scanf` para introducir cadenas de caracteres de este tipo. Para hacer esto, la conversión de caracteres tipo `s` dentro de la cadena de control es reemplazada por una secuencia de caracteres encerrados entre corchetes, designada como `[. . . .]`. Los caracteres de espaciado se pueden incluir dentro de los corchetes, pudiéndose por tanto acomodar a cadenas que contengan dichos caracteres.

Cuando se ejecuta el programa, se leerán caracteres sucesivamente del dispositivos de entrada estándar mientras cada carácter de entrada coincida con uno de los caracteres dentro de los corchetes. No es necesario que el orden de los caracteres dentro de los corchetes se corresponda con el de los caracteres que se están introduciendo. Los caracteres de entrada se pueden repetir. Sin embargo, la cadena de caracteres terminará cuando se encuentre un carácter de entrada que no coincida con ninguno de los caracteres incluidos entre los corchetes. Se añadirá entonces, automáticamente, un carácter nulo (`\0`) al final de la cadena.

EJEMPLO 4.7. Este ejemplo ilustra el uso de la función `scanf` para introducir una cadena de caracteres que consta de letras mayúsculas y espacios en blanco. La cadena será de longitud no determinada, pero estará limitada a `79` caracteres (realmente, `80` caracteres incluyendo el carácter nulo que se añade al final).

```
#include <stdio.h>
main()
{
   char linea[80];
   . . . . .
   scanf("%[ ABCDEFGHIJKLMNOPQRSTUVWXYZ]", linea);
   . . . . .
}
```

Si se introduce la cadena

```
CIUDAD DE ALMERIA
```

por el dispositivo de entrada estándar cuando se ejecuta el programa, se le asignará al array `linea` toda la cadena de caracteres, ya que está compuesta sólo por letras mayúsculas y espacios en blanco. Sin embargo, si se hubiese escrito la cadena

```
Ciudad de Almería
```

sólo se habría asignado a `linea` la letra `C`, ya que la primera letra minúscula (en este caso la `i`) se interpretaría como el primer carácter a continuación de la cadena. Se podría, por supuesto, haber incluido mayúsculas y minúsculas dentro de los corchetes, pero esto ya resulta incómodo.

Otra forma de hacer esto, que a veces es más útil, es preceder los caracteres de dentro de los corchetes por un *acento circunflejo* (^). Esto hace que los siguientes caracteres de dentro de los corchetes se interpreten de la forma opuesta. Esto es, cuando se ejecuta el programa, se leerán sucesivamente los caracteres del dispositivo de entrada estándar mientras que cada carácter de entrada *no coincida* con alguno de los caracteres incluidos en los corchetes.

Si los caracteres encerrados por los corchetes son sólo el acento circunflejo seguido de un carácter de nueva línea, la cadena de caracteres introducida por el dispositivo de entrada estándar puede tener cualquier carácter ASCII, excepto el carácter de nueva línea (*line feed*). Por tanto, el usuario puede introducir lo que desee y pulsar la tecla `INTRO`. La tecla `INTRO` proporcionará el carácter de nueva línea, indicando con ello el final de la cadena.

EJEMPLO 4.8. Supongamos que un programa en C contiene las siguientes instrucciones.

```
#include <stdio.h>
main()
{
   char linea[80];
   . . . . .
   scanf(" %[^\n]", linea);
   . . . . .
}
```

Observe el espacio en blanco delante de `%[^\n]`, para ignorar los caracteres no deseados introducidos con anterioridad.

Cuando se ejecuta la función `scanf`, se leerá del dispositivo de entrada estándar una cadena de caracteres de longitud no determinada (pero no más de 79 caracteres) y se le asignará a `linea`. No habrá restricciones en los caracteres que la compongan, salvo que quepan en una línea. Por ejemplo, la cadena

```
¡El REAL MADRID es uno de los mejores equipos de fútbol españoles!
```

se podría introducir por el teclado y se le asignaría a `linea`.

4.5. MÁS SOBRE LA FUNCIÓN `scanf`

Esta sección contiene algunos detalles adicionales de la función `scanf`. Los programadores que estén empezando ahora con C pueden saltarse esta sección por el momento, si lo desean.

Los caracteres consecutivos que no sean de espaciado y que componen un dato, definen, en conjunto, un *campo*. Es posible limitar el número de los caracteres especificando una *longitud de campo* máxima para ese dato. Para hacer esto, se introduce en la cadena de control un entero sin signo que indica la longitud de campo, entre el signo de porcentaje (`%`) y el carácter de conversión.

El dato puede estar compuesto por menos caracteres de los que especifique la longitud de campo. Sin embargo, el número de caracteres del dato real no puede exceder la longitud de campo especificada. No se leerán los caracteres que se encuentren más allá de la longitud de campo especificada. Estos caracteres sobrantes pueden ser interpretados de forma incorrecta como los componentes del siguiente dato.

EJEMPLO 4.9. A continuación se muestra la estructura esquemática de un programa en C.

```
#include <stdio.h>

main()
{
   int a, b, c;

   . . . . .

   scanf("%3d %3d %3d", &a, &b, &c);

   . . . . .
}
```

Cuando se ejecuta el programa, se introducen tres cantidades enteras por el dispositivo de entrada estándar (el teclado). Supongamos que los datos introducidos son:

```
1 2 3
```

Entonces se efectuarán las siguientes asignaciones:

```
a = 1, b = 2, c = 3
```

Si se introducen los datos

```
123 456 789
```

Entonces las asignaciones serían:

```
a = 123, b = 456, c = 789
```

Supongamos ahora que se introduce

```
123456789
```

Entonces las asignaciones serían:

```
a = 123, b = 456, c = 789
```

como antes, ya que se le asignarían a a los tres primeros dígitos, los tres siguientes dígitos a b y los últimos tres dígitos a c.

Finalmente, supongamos que los datos que se introducen son

```
1234 5678 9
```

Las asignaciones resultantes serían ahora

```
a = 123, b = 4, c = 567
```

Los otros dos dígitos restantes (8 y 9) se ignorarán, a no ser que se lean a continuación en otra instrucción scanf.

EJEMPLO 4.10. Consideremos un programa en C que contiene las siguientes instrucciones.

```
#include <stdio.h>

main()
{
    int i;
    float x;
    char c;

    . . . . .

    scanf("%3d %5f %c", &i, &x, &c);

    . . . . .
}
```

Si introducimos los siguientes datos

```
10 256.875 T
```

cuando se ejecuta el programa, entonces se le asignará a i el valor 10, a x el valor 256.8 y a c se le asignará el carácter 7. Los otros dos caracteres (5 y T) se ignorarán.

La mayoría de las versiones de C permiten que ciertos caracteres de conversión en la cadena de control sean precedidos por un *prefijo* de una sola letra, que indica la longitud del argumento correspondiente. Por ejemplo, una `l` (L minúscula) se utiliza para indicar un argumento entero largo, tanto con signo como sin signo, o un argumento de doble precisión. Análogamente se utiliza una `h` para indicar un entero corto tanto con signo como sin signo. También algunas versiones de C permiten el uso de una `L` mayúscula para indicar un `long double`.

EJEMPLO 4.11. Supongamos que aparecen las siguientes instrucciones en un programa en C.

```
#include <stdio.h>

main()

{
   short ix, iy;
   long lx, ly;
   double dx, dy;

   . . . . .

   scanf("%hd %ld %lf", &ix, &lx, &dx);

   . . . . .

   scanf("%3ho %7lx %15le", &iy, &ly, &dy);

   . . . . .

}
```

La cadena de control de la primera función `scanf` indica que el primer dato será asignado a una variable entera corta en decimal, el segundo a una variable larga en decimal y el tercero a una variable de doble precisión. La cadena de control de la segunda función `scanf` indica que el primer dato tendrá una longitud de campo máxima de tres caracteres y será asignado a una variable entera corta en octal, el segundo dato tendrá una longitud de campo máxima de siete caracteres y será asignado a una variable entera larga en hexadecimal y el tercer dato tendrá una longitud de campo máxima de 15 caracteres y se le asignará a una variable de doble precisión.

Algunas versiones de C permiten el uso de mayúsculas como caracteres de conversión para indicar enteros largos (con signo o sin signo). Es posible que se disponga de esto además del prefijo `"l"`, o reemplazando el uso del prefijo.

EJEMPLO 4.12. Consideremos de nuevo las líneas del programa en C dadas en el Ejemplo 4.11. En algunas versiones de C es posible escribir las funciones `scanf` de forma diferente, como sigue.

```
#include <stdio.h>

main()

{
   short ix, iy;
   long lx, ly;
   double dx, dy;
```

```
    . . . . .

    scanf("%hd %D %f", &ix, &lx, &dx);

    . . . . .

    scanf("%3ho %7X %15e", &iy, &ly, &dy);

    . . . . .

}
```

Observe el uso de caracteres de conversión en mayúsculas (en las funciones `scanf`) para indicar enteros largos. La interpretación de las funciones `scanf` es la misma que la del ejemplo anterior.

En la mayoría de las versiones de C es posible saltarse un dato sin que le sea asignado a una determinada variable o formación. Para hacer esto, el signo `%` es seguido por un asterisco (`*`) dentro de la cadena de control. Esto se denomina *supresión de asignación*.

EJEMPLO 4.13. Esta es una variación del Ejemplo 4.6.

```
#include <stdio.h>

main()

{
    char concepto[20];
    int num_partida;
    float coste;

    . . . . .

    scanf(" %s %*d %f", concepto, &num_partida, &coste);

    . . . . .

}
```

Observe la presencia del asterisco en el segundo grupo de caracteres.

Si los datos de entrada son

```
cremallera 12345 0.05
```

entonces a `concepto` se le asignará `cremallera` y a `coste` se le asignará `0.05`. Sin embargo, a `num_partida` no se le asignará `12345` a causa del asterisco, que se interpreta como un carácter de supresión de asignación.

La cantidad entera `12345` será leída entre los otros datos, aunque no se le asigne a la variable correspondiente.

Si la cadena de control contiene varios grupos de caracteres sin estar separados por caracteres de espaciado, se debe tener cuidado con la conversión tipo `c`. En estos casos, si aparece entre

los datos de entrada un carácter de espaciado, se interpretará como un dato más. Para saltarse estos caracteres de espaciado y leer el siguiente carácter que no sea de espaciado se debe utilizar el grupo de conversión `%1s`.

EJEMPLO 4.14. Consideremos un programa en C que contiene las siguientes líneas.

```
#include <stdio.h>

main()

{
   char c1, c2, c3;

   . . . . .

   scanf("%c%c%c", &c1, &c2, &c3);

   . . . . .

}
```

Si los datos de entrada son

```
a b c
```

(con espacios en blanco entre las letras), se efectuarán las siguientes asignaciones:

```
c1 = a,    c2 = <espacio en blanco>,    c3 = b
```

Sin embargo, si la función `scanf` se hubiese escrito de la siguiente forma

```
scanf(" %c%1s%1s", &c1, &c2, &c3);
```

entonces los mismos datos de entrada harían que se realizasen las siguientes asignaciones:

```
c1 = a,  c2 = b,  c3 = c
```

como deseábamos.

Tenga en cuenta que hay otras formas de solucionar este problema. Podríamos haber escrito la función `scanf` así:

```
scanf(" %c %c %c", &c1, &c2, &c3);
```

con espacios en blanco dentro de la cadena de control, o podríamos haber utilizado la función `scanf` original para haber escrito los datos de entrada como caracteres consecutivos sin espacios en blanco, por ejemplo `abc`.

Cuando se encuentran dentro de la cadena de control caracteres no reconocidos, se espera que aparezcan estos caracteres en los datos de entrada. Estos caracteres de entrada se leerán,

pero no se asignarán a ningún identificador. Si la entrada no coincide con los caracteres de la cadena de control, terminará la ejecución de la función scanf.

EJEMPLO 4.15. Consideremos un programa en C con las siguientes líneas.

```
#include <stdio.h>

main()

{
    int i;
    float x;

    . . . . .

    scanf("%d a %f", &i, &x);

    . . . . .

}
```

Si los datos de entrada son

```
1 a 2.0
```

se leerá el entero decimal 1 y se le asignará a i, se leerá el carácter a a continuación y se ignorará, y se leerá el valor en coma flotante 2.0 y se le asignará a x.

Por otra parte, si la entrada hubiese sido

```
1 2.0
```

la ejecución de la función scanf hubiese acabado al no encontrar el carácter esperado (a). Por tanto, a i se le habría asignado el valor 1, pero a x se le habría asignado automáticamente el valor 0.

El lector debe tener presente que hay algunas diferencias entre las características de scanf de una versión de C a otra. Los rasgos descritos anteriormente son bastante comunes y se encuentran en prácticamente todas las versiones del lenguaje. Sin embargo, puede haber pequeñas diferencias en su implementación. Además se podrán encontrar prestaciones adicionales en algunas versiones del lenguaje.

4.6. ESCRITURA DE DATOS - LA FUNCIÓN printf

Se pueden escribir datos en el dispositivo de salida estándar utilizando la función de biblioteca printf. Se puede utilizar esta función para escribir cualquier combinación de valores numéricos, caracteres sueltos y cadenas de caracteres. Es análoga a la función de entrada scanf, con la diferencia de que su propósito es visualizar datos en lugar de introducirlos en la computadora.

Esto es, la función `printf` se ocupa de transferir datos de la memoria de la computadora al dispositivo de salida estándar, mientras que la función `scanf` introduce datos del dispositivo de entrada estándar y los almacena en la memoria de la computadora.

En términos generales, la función `printf` se escribe

```
printf(cadena de control, arg1, arg2,. . . , argn)
```

en donde *cadena de control* hace referencia a una cadena de caracteres que contiene información sobre el formato de la salida y *arg1*, *arg2*, . . ., *argn* son argumentos que representan los datos de salida. Los argumentos pueden ser constantes, variables simples o nombres de arrays o expresiones más complicadas. También se pueden incluir referencias a funciones. En contraste con la función `scanf` tratada en la sección anterior, los argumentos de la función `printf` *no* representan direcciones de memoria y por tanto no son precedidos de ampersands.

La cadena de control está compuesta por grupos de caracteres, con un grupo de caracteres por cada dato de salida. Cada grupo de caracteres debe comenzar por un signo de porcentaje (`%`). En su forma más sencilla, un grupo de caracteres consistirá en el signo de porcentaje seguido de un *carácter de conversión* que indica el tipo del dato correspondiente.

Pueden incluirse varios grupos de caracteres seguidos o separados por otros caracteres, incluidos los de espaciado. Estos «otros» caracteres son transferidos directamente al dispositivo de salida estándar, en donde son visualizados. Es muy común el uso de espacios en blanco como separadores de grupos de caracteres.

La Tabla 4.2 contiene una lista de los caracteres de conversión más frecuentemente utilizados.

Tabla 4.2. Caracteres de conversión de los datos de salida de uso común

Carácter de conversión	Significado
`c`	el dato es visualizado como un carácter
`d`	el dato es visualizado como un entero decimal con signo
`e`	el dato es visualizado como un valor en coma flotante con exponente
`f`	el dato es visualizado como un valor en coma flotante sin exponente
`g`	el dato es visualizado como un valor en coma flotante utilizando la conversión tipo `e` o tipo `f` según sea el caso. No se visualizan ni los ceros finales ni el punto decimal cuando no es necesario.
`i`	el dato es visualizado como un entero con signo
`o`	el dato es visualizado como un entero octal, sin el cero inicial
`s`	el dato es visualizado como una cadena de caracteres
`u`	el dato es visualizado como un entero decimal sin signo
`x`	el dato es visualizado como un entero hexadecimal sin el prefijo `0x`

Observe que algunos de estos caracteres tienen un significado distinto que en la función `scanf` (ver Tabla 4.1).

EJEMPLO 4.16. Aquí tenemos un programa sencillo en el que se utiliza la función `printf`.

```
#include <stdio.h>
#include <math.h>

main()          /* escribir varios números en coma flotante */
{
   float i = 2.0, j = 3.0;
   printf("%f %f %f %f", i, j, i+j, sqrt(i+j));
}
```

Observe que los dos primeros argumentos dentro de la función `printf` son variables simples, el tercer argumento es una expresión aritmética y el último argumento una referencia a una función que tiene una expresión numérica como argumento.

La ejecución del programa produce la siguiente salida:

```
2.000000 3.000000 5.000000 2.236068
```

EJEMPLO 4.17. Supongamos que tenemos un programa en C que incluye las siguientes líneas. En ellas puede verse cómo se pueden visualizar distintos tipos de datos mediante la función `printf`.

```
#include <stdio.h>

main()
{
   char concepto[20];
   int num_partida;
   float coste;

   . . . . .

   printf("%s %d %f", concepto, no_partida, coste);

   . . . . .
}
```

La cadena de control de la función `printf` es `"%s %d %f"`. Contiene tres grupos de caracteres. El primer grupo de caracteres, `%s`, indica que el primer argumento (`concepto`) representa una cadena de caracteres. El segundo grupo de caracteres, `%d`, indica que el segundo argumento (`num_partida`) representa un valor entero decimal, y el tercer grupo de caracteres, `%f`, indica que el tercer argumento (`coste`) representa un valor en coma flotante.

Observe que los argumentos no están precedidos por ampersands. Esto es una variación sobre la función `scanf`, que requiere ampersands en todos los argumentos que no sean nombres de formaciones (ver Ejemplo 4.5).

Supongamos ahora que a `concepto`, `num_partida` y `coste` se les ha asignado los valores `cremallera`, `12345` y `0.05`, respectivamente, dentro del programa. Cuando se ejecuta la instrucción `printf`, se genera la siguiente salida.

```
cremallera 12345 0.050000
```

Los espacios en blanco entre los datos son generados por los espacios en blanco que aparecen en la cadena de control de la instrucción printf.

Supongamos que la instrucción printf se ha escrito así:

```
printf("%s%d%f", concepto, num_partida, coste);
```

La instrucción printf es válida sintácticamente, aunque hace que los datos de salida se presenten sin separación, esto es,

```
cremallera123450.050000
```

La conversión tipo f y la tipo e se utilizan para visualizar valores en coma flotante. Sin embargo, la última hace que se incluya en la salida el exponente, mientras que la primera no.

EJEMPLO 4.18. El siguiente programa genera la salida del mismo valor en coma flotante de dos formas distintas.

```
#include <stdio.h>

main()    /* visualizar un valor en coma flotante de dos
             formas diferentes */
{
   double x = 5000.0, y = 0.0025;

   printf("%f %f %f %f\n\n", x, y, x*y, x/y);
   printf("%e %e %e %e", x, y, x*y, x/y);
}
```

Las dos instrucciones printf tienen los mismos argumentos. Sin embargo, la primera instrucción printf utiliza la conversión tipo f, mientras que la segunda utiliza la conversión tipo e. Observe también la repetición del carácter de nueva línea en la primera instrucción printf. Esto hace que la salida aparezca doblemente espaciada, como se muestra a continuación.

Cuando se ejecuta el programa, se genera la siguiente salida.

```
5000.000000 0.002500 12.500000 2000000.000000

5.000000e+03 2.500000e-03 1.250000e+01 2.000000e+06
```

La primera línea de salida muestra las cantidades representadas por x, y, x*y y x/y en formato en coma flotante estándar, sin exponentes. La segunda línea de salida muestra las mismas cantidades en notación científica, con exponentes.

Observe que cada valor se presenta con seis cifras decimales. Se puede modificar el número de cifras decimales especificando la *precisión* en cada grupo de caracteres dentro de la cadena de control (hay más información sobre esto en la sección 4.7).

La función printf interpreta la conversión tipo s de forma diferente a como lo hace la función scanf. En la función printf se utiliza la conversión tipo s para visualizar una cadena de caracteres que acaba en el carácter nulo (\0). Se pueden incluir caracteres de espaciado en la cadena de caracteres.

EJEMPLO 4.19. Lectura y escritura de una línea de texto. El siguiente es un pequeño programa en C que lee una línea de texto y después la escribe, tal y como se introdujo. El programa ilustra las diferencias sintácticas en la lectura y la escritura de una cadena de caracteres que contiene, entre otros, caracteres de espaciado.

```
#include <stdio.h>

main()    /* leer y escribir una línea de texto */
{
   char linea[80];
   scanf(" %[^\n]", linea);
   printf("%s", linea);
}
```

Observe la diferencia en las cadenas de control de las funciones `scanf` y `printf`.

Supongamos ahora que se introduce por el dispositivo de entrada estándar la siguiente cadena de caracteres cuando se ejecuta el programa.

```
¡El REAL MADRID es uno de los mejores equipos de fútbol españoles!
```

Esta cadena de caracteres contiene letras minúsculas, mayúsculas, signos de puntuación y caracteres de espaciado. Se puede introducir la cadena completa con la función `scanf` simplemente, siempre que termine en un carácter de nueva línea (pulsando la tecla `INTRO`). La función `printf` hará que se visualice la cadena completa en el dispositivo de salida estándar, tal y como se introdujo. Por tanto, la computadora generará el mensaje

```
¡El REAL MADRID es uno de los mejores equipos de fútbol españoles!
```

Se puede especificar una longitud de campo *mínima* anteponiendo al carácter de conversión un entero sin signo. Si el número de caracteres del dato correspondiente es menor que la longitud de campo especificada, entonces el dato será precedido por los espacios en blanco necesarios para que se consiga la longitud de campo especificada. Si el número de caracteres del dato excede la longitud de campo especificada, se visualizará el dato completo. Este comportamiento es justo el contrario al del indicador de longitud de campo en la función `scanf`, que especifica una longitud de campo *máxima*.

EJEMPLO 4.20. El siguiente programa en C ilustra el uso del indicador de longitud de campo mínima.

```
#include <stdio.h>

main()    /* especificación de longitud de campo mínima */

{
   int i = 12345;
   float x = 345.678;

   printf("%3d %5d %8d\n\n", i, i, i);
   printf("%3f %10f %13f\n\n", x, x, x);
   printf("%3e %13e %16e", x, x, x);
}
```

Observe la aparición de dos caracteres de nueva línea seguidos en las dos primeras instrucciones `printf`. Esto hará que las líneas de salida aparezcan con separación doble, como se muestra a continuación.

Cuando se ejecuta el programa, se genera la siguiente salida.

```
12345 12345    12345

345.678000 345.678000   345.678000

3.456780e+02   3.456780e+02      3.456780e+02
```

En la primera línea de salida aparece un entero en decimal utilizando tres longitudes de campo mínimas (tres, cinco y ocho caracteres). El valor entero completo se visualiza dentro de cada campo, aunque la longitud de campo sea demasiado pequeña (como ocurre con el primer campo en este ejemplo).

El segundo valor de salida de la primera línea es precedido por un espacio en blanco. Este espacio en blanco es generado por el que aparece en la cadena de control separando los dos primeros grupos de caracteres.

El tercer valor de salida es precedido por cuatro espacios en blanco. Un espacio en blanco se debe al que aparece en la cadena de control separando los dos últimos grupos de caracteres. Los otros tres espacios en blanco rellenan la longitud de campo mínima, que es mayor que el número de caracteres del valor de salida (la longitud de campo mínima es ocho, pero el valor de salida tiene sólo cinco caracteres).

Se pueden observar situaciones análogas en las dos líneas siguientes, en las que se visualiza un valor en coma flotante utilizando la conversión tipo `f` (línea 2) y la conversión tipo `e` (línea 3).

EJEMPLO 4.21. Presentamos una variación del programa del Ejemplo 4.20, que utiliza la conversión tipo `g`.

```
#include <stdio.h>

main()          /* especificación de longitud de campo mínima */
{
   int i = 12345;
   float x = 345.678;

   printf("%3d %5d %8d\n\n", i, i, i);
   printf("%3g %10g %13g\n\n", x, x, x);
   printf("%3g %13g %16g", x, x, x);
}
```

La ejecución de este programa hace que se visualicen las siguientes líneas.

```
12345 12345   12345

345.678   345.678         345.678

345.678       345.678         345.678
```

Los valores en coma flotante se visualizan con una conversión tipo `f`, lo que conlleva una escritura más corta. Las longitudes de campo se adecúan a las especificaciones que se hacen en la cadena de control.

4.7. MÁS SOBRE LA FUNCIÓN `printf`

Esta sección contiene detalles adicionales sobre la función `printf`. Los programadores que estén comenzando con C pueden saltarse esta sección por el momento, si lo desean.

Ya hemos aprendido cómo especificar una longitud de campo mínima en la función `printf`. También es posible especificar el máximo número de cifras decimales para un valor en coma flotante, o el máximo número de caracteres para una cadena de caracteres. Esta especificación se denomina *precisión*. La precisión es un entero sin signo que siempre es precedido por un punto decimal. Si se especifica la longitud de campo mínima además de la precisión (como de hecho suele hacerse), entonces la especificación de la precisión sigue a la especificación de la longitud de campo. Estas dos especificaciones enteras preceden al carácter de conversión.

Un número en coma flotante se *redondeará* si se debe recortar para ajustarse a la precisión especificada.

EJEMPLO 4.22. A continuación se muestra un programa que ilustra el uso de la especificación de la precisión con números en coma flotante.

```
#include <stdio.h>

main()  /* visualizar un número en coma flotante
           con varias precisiones diferentes */
{
   float x = 123.456;

   printf("%7f %7.3f %7.1f\n\n", x, x, x);
   printf("%12e %12.5e %12.3e", x, x, x);
}
```

Cuando se ejecuta el programa, se genera la siguiente salida:

```
123.456000 123.456   123.5

1.234560e+02 1.23456e+02   1.235e+02
```

La primera línea es producida por la conversión tipo `f`. Observe que el tercer número se ha redondeado a causa de la precisión especificada (una cifra decimal). Observe también que se añaden espacios en blanco para conseguir una longitud de campo mínima especificada (siete caracteres).

La segunda línea, producida por una conversión tipo `e`, tiene análogas características. De nuevo vemos que se ha redondeado el tercer número para satisfacer la especificación de precisión hecha (tres cifras decimales). Observe también los espacios en blanco añadidos para satisfacer la longitud de campo mínima (12 caracteres).

No es necesario que la especificación de precisión vaya acompañada de la longitud de campo mínima. Es posible especificar la precisión sin la longitud de campo mínima, aunque la precisión debe seguir apareciendo precedida por un punto decimal.

EJEMPLO 4.23. Reescribamos ahora el programa mostrado en el último ejemplo sin ninguna especificación de longitud de campo mínima, pero con especificaciones de precisión.

```
#include <stdio.h>

main()    /* visualizar un número en coma flotante
             con varias precisiones diferentes */

{
   float x = 123.456;
   printf("%f %.3f %.1f\n\n", x, x, x);
   printf("%e %.5e %.3e", x, x, x);
}
```

La ejecución de este programa produce la salida siguiente.

```
123.456000 123.456 123.5

1.234560e+02 1.23456e+02 1.235e+02
```

Observe que el tercer número de cada línea no es precedido por varios espacios en blanco, ya que no hay longitud de campo mínima que satisfacer. Salvo en este caso, la salida es la misma que la generada en el ejemplo anterior.

Las especificaciones de longitud de campo mínima y de precisión se pueden aplicar a datos de tipo carácter además de a datos numéricos. Cuando se aplica a una cadena de caracteres, la longitud de campo mínima se interpreta de la misma forma que con una cantidad numérica; es decir, se añaden espacios en blanco si la cadena de caracteres es más corta que la longitud de campo mínima especificada, y se presentará la cadena de caracteres completa si es más larga que la longitud de campo mínima especificada. Por tanto, la especificación de la longitud de campo mínima no impedirá nunca que se visualice la cadena de caracteres completa.

Sin embargo, la especificación de la precisión determina el máximo número de caracteres que pueden ser visualizados. Si la precisión especificada es menor que el número total de caracteres de la cadena, no se mostrarán los caracteres sobrantes de la parte derecha de la cadena. Esto ocurrirá aun cuando la longitud de campo mínima sea mayor que la cadena de caracteres, añadiéndose en este caso caracteres en blanco a la cadena truncada.

EJEMPLO 4.24. Las siguientes líneas de programa ilustran el uso de las especificaciones de longitud de campo mínima y de precisión en la visualización de una cadena de caracteres.

```
#include <stdio.h>

main()
{
   char linea[12];

   . . . . .

   printf("%10s %15s %15.5s %.5s", linea, linea, linea, linea);
}
```

Supongamos ahora que se le asigna a `linea` la cadena de caracteres `hexadecimal`. Cuando el programa se ejecuta, se genera la siguiente salida:

```
hexadecimal      hexadecimal          hexad hexad
```

La primera cadena de caracteres se visualiza completa, aunque esta cadena consta de `11` caracteres y la longitud de campo mínima es sólo de `10` caracteres. De esta forma, la primera cadena sobrepasa la especificación de longitud de campo mínima. A la segunda cadena de caracteres se añaden cuatro espacios en blanco a la izquierda para satisfacer la longitud de campo mínima de `15` caracteres; de esta forma, se dice que la segunda cadena está *ajustada a la derecha* de su campo. La tercera cadena consta de sólo cinco caracteres que no son de espaciado, a causa de la especificación de la precisión de cinco caracteres; sin embargo, se han añadido `10` espacios en blanco para satisfacer la especificación de longitud de campo mínima, que es de `15` caracteres. La última cadena también consta de cinco caracteres que no son de espaciado. No se añaden espacios en blanco porque no se ha especificado ninguna longitud de campo mínima.

La mayoría de las versiones de C permiten el uso de prefijos dentro de la cadena de control para indicar la longitud del argumento correspondiente. Los prefijos permitidos son los mismos que los prefijos utilizados con la función `scanf`. Así, una `l` (L minúscula) indica un argumento entero largo con o sin signo, o un argumento de doble precisión; una `h` indica un entero corto con o sin signo. Algunas versiones de C permiten una `L` (mayúscula) para indicar un `long double`.

EJEMPLO 4.25. Supongamos que un programa en C incluye las siguientes instrucciones.

```
#include <stdio.h>

main()
{
   short a, b;
   long c, d;

   . . . . .

   printf("%5hd %6hx %8lo %lu", a, b, c, d);

   . . . . .
}
```

La cadena de control indica que el primer dato será un entero corto en decimal, el segundo un entero corto en hexadecimal, el tercero un entero largo octal y el cuarto un entero largo sin signo (decimal). Nótese que los tres primeros grupos tienen especificada la longitud de campo mínima, pero el cuarto no.

Algunas versiones de C permiten que se escriban en mayúsculas los caracteres de conversión `X`, `E` y `G`. Estos caracteres de conversión en mayúsculas hacen que cualquier letra de los datos de salida se presente en mayúsculas. (Nótese que este uso de caracteres de conversión en mayúsculas es diferente que el que se hace en la función `scanf`.)

EJEMPLO 4.26. El siguiente programa ilustra la utilización de caracteres de conversión en mayúsculas en la función `printf`.

```
#include <stdio.h>

main()     /* uso de caracteres de conversión en mayúsculas */
{
   int a = 0x80ec;
   float b = 0.3e-12;

   printf("%4x %10.2e\n\n", a, b);
   printf("%4X %10.2E", a, b);
}
```

Observe que la primera instrucción `printf` contiene caracteres de conversión en minúsculas, mientras que la segunda instrucción `printf` contiene caracteres de conversión en mayúsculas.

Cuando se ejecuta el programa, se genera la siguiente salida.

```
80ec   3.00e-13

80EC   3.00E-13
```

La primera cantidad en cada línea es un número hexadecimal. Nótese que las letras `ec` (que son parte del número hexadecimal) se muestran en minúsculas en la primera línea y en mayúsculas en la segunda.

La segunda cantidad de cada línea es un número decimal en coma flotante que incluye un exponente. Observe que la letra `e`, que indica el exponente, se muestra como minúscula en la primera línea y como mayúscula en la segunda.

Se le recuerda de nuevo al lector que no todos los compiladores permiten el uso de caracteres de conversión en mayúsculas.

Además de la longitud de campo mínima, la precisión y el carácter de conversión, cada grupo de caracteres dentro de la cadena de control puede incluir un *indicador*, que afecta a cómo se muestra la salida. El indicador se debe poner inmediatamente a continuación del signo de porcentaje (`%`). Algunos compiladores permiten incluir dos o más indicadores seguidos dentro del mismo grupo de caracteres. La Tabla 4.3 muestra los indicadores de uso más común.

EJEMPLO 4.27. A continuación se presenta un sencillo programa en C que ilustra el uso de indicadores con cantidades enteras y en coma flotante.

```
#include <stdio.h>

main() /* uso de indicadores con números enteros y en coma flotante */
{
   int i = 123;
   float x = 12.0, y = -3.3;

   printf(":%6d %7.0f %10.1e:\n\n", i, x, y);
   printf(":%-6d %-7.0f %-10.1e:\n\n", i, x, y);
   printf(":%+6d %+7.0f %+10.1e:\n\n", i, x, y);
   printf(":%-+6d %-+7.0f %-+10.1e:\n\n", i, x, y);
   printf(":%7.0f %#7.0f %7g %#7g:", x, x, y, y);
}
```

Tabla 4.3. Indicadores de uso común

Indicador	Significado
-	El dato se ajusta a la izquierda dentro del campo (si se requieren espacios en blanco para conseguir la longitud de campo mínima, se añaden *después* del dato en lugar de *antes*).
+	Cada dato numérico es precedido por un signo (+ o –). Sin este indicador, sólo los datos negativos son precedidos por el signo –.
0	Hace que se presenten ceros en lugar de espacios en blanco. Se aplica sólo a datos que estén ajustados a la derecha dentro de campos de longitud mayor que la del dato. (*Nota:* Algunos compiladores consideran el indicador cero como parte de la especificación de la longitud de campo en lugar de como indicador real. Esto asegura que el 0 es procesado el último si hay varios indicadores presentes.)
' '	(*espacio en blanco*) Cada dato numérico positivo es precedido por un espacio en blanco. Este indicador se ignora si se encuentra presente también el indicador +.
#	(*con las conversiones tipo* o *y tipo* x) Hace que los datos octales y hexadecimales sean precedidos por 0 y 0x, respectivamente.
#	(*con las conversiones tipo* e, *tipo* f *y tipo* g) Hace que se presenten todos los números en coma flotante con un punto, aunque tengan un valor entero. Impide el truncamiento de los ceros de la derecha realizada por la conversión tipo g.

Cuando se ejecuta el programa, se produce la siguiente salida. (Los dos puntos indican el principio del primer campo y el final del último campo de cada línea.)

```
:   123      12   -3.3e+00:

:123     12       -3.3e+00  :

:     +123   +12  -3.3e+00:

:+123    +12      -3.3e+00  :

:     12     12.      -3.3 -3.30000:
```

La primera línea sirve para ilustrar cómo aparecen los números enteros y en coma flotante sin ningún indicador. Cada número es ajustado a la derecha dentro de su respectivo campo. La segunda línea muestra los mismos números, utilizando las mismas conversiones, con un indicador - en cada grupo de caracteres. Observe que los números se ajustan ahora a la izquierda dentro de sus campos respectivos. La tercera línea muestra el efecto de utilizar un indicador +. Los números ahora se ajustan a la derecha, como en la primera línea, pero cada número (tanto positivos como negativos) es precedido ahora por su signo.

En la cuarta línea vemos el efecto de combinar un indicador - con un indicador +. Los números están ahora ajustados a la izquierda y precedidos por su signo. Finalmente, la última línea muestra dos números en coma flotante, cada uno mostrado primero sin el indicador #, y después con él. Observe que el efecto

del indicador es incluir un punto decimal en el número `12.` (que es escrito con la conversión de tipo `f`) e incluir la ristra de ceros en el número `-3.30000` (escrito con la conversión tipo `g`).

EJEMPLO 4.28. Consideremos ahora el siguiente programa, que visualiza números decimales, octales y hexadecimales.

```
#include <stdio.h>

main() /* uso de indicadores con números sin signo
          decimales, octales y hexadecimales */
{
   int i = 1234, j = 01777, k =0xa08c;

   printf(":%8u %8o %8x:\n\n", i, j, k);
   printf(":%-8u %-8o %-8x:\n\n", i, j, k);
   printf(":%#8u %#8o %#8X:\n\n", i, j, k);
   printf(":%08u %08o %08X:\n\n", i, j, k);
}
```

La ejecución de este programa origina la siguiente salida. (Los dos puntos indican el comienzo del primer campo y el final del último campo de cada línea.)

```
:    1234     1777     a08c:

:1234     1777     a08c    :

:    1234    01777   0XA08C:

:00001234 00001777 0000A08C:
```

La primera línea muestra la visualización de un entero sin signo, octal y hexadecimal sin indicadores. Observe que los números están ajustados a la derecha dentro de sus respectivos campos. La segunda línea muestra los mismos datos, utilizando las mismas conversiones, pero con el indicador `-` en cada grupo de caracteres. Ahora los números se encuentran ajustados a la izquierda dentro de sus respectivos campos.

En la tercera línea vemos la salida que se obtiene cuando se utiliza el indicador `#`. Este indicador hace que el número octal `1777` sea precedido por un `0` (apareciendo como `01777`) y el número hexadecimal aparezca precedido por `0X` (esto es, `0XA08C`). Observe que el entero sin signo decimal `1234` no se ve afectado por este indicador. Observe también que el número hexadecimal contiene ahora letras mayúsculas, ya que el carácter de conversión está escrito como mayúscula `(X)`.

La última línea ilustra el uso del indicador `0`. Este indicador hace que los campos se rellenen con ceros en lugar de espacios en blanco. Vemos de nuevo caracteres hexadecimales en mayúsculas, en respuesta al carácter de conversión escrito como mayúscula `(X)`.

EJEMPLO 4.29. El siguiente esquema de programa sirve como ejemplo del uso de indicadores en la salida de cadenas de caracteres.

```
#include <stdio.h>

main()
{
   char linea[12];
```

```
    . . . . .
    printf(":%15s %15.5s %.5s:\n\n", linea, linea, linea);
    printf(":%-15s %-15.5s %-.5s", linea, linea, linea);
}
```

Supongamos ahora que al array de caracteres `linea` se le asigna la cadena de caracteres `menor-caso`. Se generará la siguiente salida cuando se ejecute el programa.

```
:      menor-caso          menor menor:

:menor-caso      menor           menor:
```

En la primera línea se ve cómo se visualizan las cadenas de caracteres cuando no se incluyen indicadores, como se explicó en el Ejemplo 4.24. La segunda línea muestra las mismas cadenas, ajustadas a la izquierda, en respuesta al indicador - en cada grupo de caracteres.

Los caracteres dentro de la cadena de control que no se reconozcan se visualizarán tal y como aparezcan. Esto nos permite incluir rótulos y mensajes entre los datos, si así lo deseamos.

EJEMPLO 4.30. En el siguiente programa se muestra cómo se pueden visualizar rótulos.

```
#include <stdio.h>

main()   /* salida de números en coma flotante con rótulos */
{
    float a = 2.2, b = -6.2, x1 = .005, x2 = -12.88;

    printf("$%4.2f   %7.1f%%\n\n", a, b);
    printf("x1=%7.3f    x2=%7.3f", x1, x2);
}
```

Este programa hace que el valor de `a (2.2)` sea precedido por el signo del dólar `($)` y que se presente tras el valor de `b (-6.2)` un signo de porcentaje `(%)`. Observe los dos signos de porcentaje consecutivos en la primera instrucción `printf`. El primer signo de porcentaje indica el comienzo de un grupo de caracteres, mientras que el segundo es interpretado como un rótulo.

La segunda instrucción `printf` hace que el valor de `x1` aparezca precedido del rótulo `x2=`. Aparecerán tres espacios en blanco separando estos datos rotulados.

Se muestra a continuación la salida generada.

```
$2.20     -6.2%

x1=1 0.005    x2=-12.880
```

El lector debe tener presentes las posibles variaciones de las características de la función `printf` en diferentes versiones de C. Las descritas anteriormente en esta sección son muy comunes, aunque pueden existir algunas variaciones en la forma en que se realicen. También se suele disponer de otras posibilidades en muchas versiones del lenguaje.

4.8. LAS FUNCIONES gets Y puts

C dispone de un cierto número de funciones de biblioteca que permiten diferentes formas de transferencia de datos hacia dentro o fuera de la computadora. Veremos algunas de estas funciones en el Capítulo 12, en el que trataremos los archivos de datos. Sin embargo, antes de terminar este capítulo mencionaremos las funciones `gets` y `puts`, que facilitan la transferencia de cadenas de caracteres entre la computadora y los dispositivos de entrada/salida estándar.

Cada una de estas funciones acepta un solo argumento. El argumento debe ser un dato que represente una cadena de caracteres (una formación de caracteres). La cadena de caracteres puede incluir caracteres de espaciado. En el caso de `gets`, la cadena se introducirá por el teclado y terminará con un carácter de nueva línea (por ejemplo la cadena terminará cuando el usuario pulse la tecla `Intro`).

Las funciones `gets` y `puts` ofrecen alternativas sencillas al uso de `scanf` y `printf` para la lectura y escritura de cadenas de caracteres, como se muestra en el siguiente ejemplo.

EJEMPLO 4.31. **Lectura y escritura de una línea de texto.** Ésta es otra versión del programa presentado en el Ejemplo 4.19, que lee una línea de texto y después la escribe en su forma original.

```
#include <stdio.h>

main()    /* leer y escribir una línea de texto */

{
   char linea[80];

   gets(linea);
   puts(linea);
}
```

Este programa utiliza `gets` y `puts` en lugar de `scanf` y `printf` para transferir texto hacia dentro y hacia fuera de la computadora. Nótese que la sintaxis es más sencilla en este programa (compárese con el programa del Ejemplo 4.19). Por otra parte, las funciones `scanf` y `printf` en el programa anterior se podrían expandir incluyendo datos adicionales, mientras que en el presente programa no se puede.

Cuando se ejecuta este programa, se comporta de la misma forma que el mostrado en el Ejemplo 4.19.

4.9. PROGRAMACIÓN INTERACTIVA (CONVERSACIONAL)

Muchos programas de computadora modernos están diseñados para crear un diálogo interactivo entre la computadora y la persona que utiliza el programa (el «usuario»). Estos diálogos suelen involucrar una interacción de tipo pregunta-respuesta, en la que la computadora hace las preguntas y el usuario proporciona las respuestas, o viceversa. De esta forma parece que la computadora y el usuario están desarrollando una cierta forma de conversación limitada.

En C estos diálogos se pueden crear mediante el uso alternativo de las funciones `scanf` y `printf`. La programación real es sencilla, aunque a veces causa confusión entre los programadores que se inician en C que la función `printf` se utilice cuando se introducen datos (para

visualizar las preguntas de la computadora) y cuando se muestran resultados. Por otra parte, `scanf` sólo se utiliza para la verdadera entrada de datos.

En el siguiente ejemplo se plasman las ideas básicas.

EJEMPLO 4.32. Calificaciones medias de exámenes. En este ejemplo se presenta un sencillo programa interactivo en C que lee el nombre de un estudiante y tres calificaciones de exámenes, y después calcula la calificación media. Los datos se introducen de forma interactiva, la computadora pide la información al usuario y éste la proporciona con un formato libre. La entrada de cada dato se hace en una línea. Una vez que se han introducido todos los datos, la computadora calculará la media deseada y escribirá todos los datos (tanto los de entrada como la media calculada).

A continuación se presenta el programa.

```
#include <stdio.h>

main()                          /* ejemplo de programa interactivo */

{
    char nombre[20];
    float nota1, nota2, nota3, media;

    printf("Por favor, introduce tu nombre: ");
                                            /* introducir nombre */
    scanf(" %[^\n]", nombre);

    printf("Por favor, introduce la primera nota: ");
                                            /* introducir nota 1 */
    scanf("%f", &nota1);

    printf("Por favor, introduce la segunda nota: ");
                                            /* introducir nota 2 */
    scanf("%f", &nota2);

    printf("Por favor, introduce la tercera nota: ");
                                            /* introducir nota 3 */
    scanf("%f", &nota3);

    media = (nota1+nota2+nota3)/3;            /*calcular la media */

    printf("\n\nNombre: %-s\n\n", nombre);     /*escribir salida */
    printf("Nota 1: %-5.1f\n", nota1);
    printf("Nota 2: %-5.1f\n", nota2);
    printf("Nota 3: %-5.1f\n\n", nota3);
    printf("Media: %-5.1f\n\n", media);
}
```

Observe que hay dos instrucciones asociadas a cada dato de entrada. La primera es una instrucción `printf`, que genera la solicitud del dato. La segunda instrucción, una función `scanf`, hace que se introduzca el dato del dispositivo de entrada estándar (por ejemplo el teclado).

Después del nombre del estudiante y las tres calificaciones de los exámenes se calcula la puntuación media. A continuación se visualizan los datos de entrada y la media calculada como resultado de la ejecución del grupo de instrucciones `printf` que aparecen al final del programa.

Se muestra a continuación una sesión interactiva típica. Se han subrayado las respuestas del usuario para ilustrar la naturaleza del diálogo.

```
Por favor, introduce tu nombre: Rosa Ramírez
Por favor, introduce la primera nota: 88
Por favor, introduce la segunda nota: 62.5
Por favor, introduce la tercera nota: 90

Nombre: Rosa Ramírez

Nota 1:     88.0
Nota 2:     62.5
Nota 3:     90.0

Media :     80.2
```

En muchos ejemplos de los capítulos siguientes de este libro se podrán ver otros programas interactivos.

CUESTIONES DE REPASO

4.1. ¿Cuáles son las funciones de entrada/salida más comúnmente usadas en C? ¿Cómo se accede a ellas?

4.2. ¿Cómo se llama el archivo de cabecera de entrada/salida estándar en la mayoría de las versiones de C? ¿Cómo se puede incluir el archivo en un programa?

4.3. ¿Cuál es el propósito de la función `getchar`? ¿Cómo se utiliza dentro de un programa en C?

4.4. ¿Qué ocurre cuando se da la condición de fin de archivo al leer caracteres con la función `getchar`? ¿Cómo se reconoce la condición de fin de archivo?

4.5. ¿Cómo se puede utilizar la función `getchar` para leer cadenas de caracteres?

4.6. ¿Cuál es el propósito de la función `putchar`? ¿Cómo se utiliza dentro de un programa en C? Compararla con la función `getchar`.

4.7. ¿Cómo se puede utilizar la función `putchar` para escribir cadenas de caracteres?

4.8. ¿Qué es un array de tipo carácter? ¿Qué representa cada elemento de un array de tipo carácter? ¿Cómo se utilizan los arrays de tipo carácter para representar cadenas de caracteres?

4.9. ¿Cuál es el propósito de la función `scanf`? ¿Cómo se utiliza dentro de un programa en C? Compararla con la función `getchar`.

4.10. ¿Cuál es el propósito de la cadena de control en la función `scanf`? ¿Qué tipo de información aporta? ¿De qué está compuesta la cadena de control?

4.11. ¿Cómo se identifica cada grupo de caracteres dentro de la cadena de control? ¿Cuáles son los caracteres constituyentes de un grupo de caracteres?

4.12. Si una cadena de control dentro de una función `scanf` contiene varios grupos de caracteres, ¿cómo se separan los grupos de caracteres? ¿Se requieren caracteres de espaciado?

4.13. Si existen caracteres de espaciado dentro de una cadena de control, ¿cómo se interpretan?

4.14. Comentar el significado de los caracteres de conversión utilizados más frecuentemente en la cadena de control de la función `scanf`.

4.15. ¿Qué símbolos especiales se deben incluir con los argumentos, además de la cadena de control, en la función `scanf`? ¿En qué se diferencia el tratamiento de los arrays de los restantes argumentos?

4.16. Cuando se introducen datos mediante la función `scanf`, ¿qué relaciones debe haber entre los datos y los argumentos correspondientes? ¿Cómo se separan varios datos entre sí?

4.17. Cuando se introducen datos mediante la función `scanf`, ¿deben ir precedidos los datos octales de `0`? ¿Deben ir precedidos de `0x` (o `0X`) los datos hexadecimales? ¿Cómo se deben escribir los datos en coma flotante?

4.18. Cuando se introduce una cadena de caracteres mediante la función `scanf` utilizando una conversión tipo `s`, ¿cómo finaliza la cadena?

4.19. Cuando se introduce una cadena de caracteres mediante la función `scanf`, ¿cómo se puede introducir una cadena que contenga caracteres de espaciado?

4.20. Idear un método para la introducción de cadenas de caracteres de longitud indeterminada que contenga caracteres de espaciado y todos los caracteres imprimibles, y que termine al pulsar el retorno de carro. Reponder a la cuestión de forma relativa al tipo de conversión requerido en la cadena de control de una función `scanf`.

4.21. ¿Qué se entiende por campo?

4.22. ¿Cómo se puede especificar la longitud de campo máxima para un dato dentro de la función `scanf`?

4.23. ¿Qué ocurre si un dato de entrada contiene más caracteres que la longitud de campo máxima fijada? ¿Y si el dato tiene menos caracteres?

4.24. ¿Cómo se pueden indicar los argumentos enteros cortos, enteros largos y de doble precisión en la cadena de control de la función `scanf`?

4.25. ¿Cómo se pueden indicar los argumentos `long double` dentro de la cadena de control de la función `scanf`? ¿Se dispone de esta posibilidad en la mayoría de las versiones de C?

4.26. ¿Cómo se puede suprimir la asignación de un dato de entrada al argumento correspondiente?

4.27. Si la cadena de control de una función `scanf` contien varios grupos de caracteres que no están separados por caracteres de espaciado, ¿qué problema puede aparecer al utilizar la conversión tipo `c`? ¿Cómo se puede solucionar esto?

4.28. ¿Cómo se interpretan los caracteres no reconocidos dentro de la cadena de control de la función `scanf`?

4.29. ¿Cuál es el propósito de la función `printf`? ¿Cómo se utiliza en un programa en C? Compárese con la función `scanf`.

4.30. ¿En qué difiere la cadena de control de la función `printf` de la de la función `scanf`?

4.31. Si la cadena de control de la función `printf` contiene varios grupos de caracteres, ¿cómo se pueden separar los grupos de caracteres? ¿Cómo se interpretan los separadores?

4.32. Citar los caracteres de conversión de uso más frecuente en la cadena de control de la función `printf` y explicar su significado. Compárense con los caracteres de conversión que se utilizan en la función `scanf`.

4.33. En la función `printf`, ¿deben ir precedidos los argumentos (aquellos que no sean cadenas de caracteres) de ampersands? Compárese con la función `scanf` y explíquense las diferencias.

4.34. ¿Cuál es la diferencia entre las conversiones tipo `f`, tipo `e` y tipo `g` cuando se están presentando datos mediante la función `printf`?

4.35. Comparar el uso de la conversión tipo `s` en las funciones `printf` y `scanf`. ¿En qué difiere la conversión tipo `s` cuando se están tratando cadenas de caracteres que contienen caracteres de espaciado?

4.36. ¿Cómo se puede especificar en la función `printf` la longitud de campo mínima para un dato?

4.37. ¿Qué ocurre si un dato contiene más caracteres que los especificados en la longitud de campo mínima? ¿Y si el dato contiene menos caracteres? Compárese con la especificación de longitud de campo máxima en la función `scanf`.

4.38. ¿Qué se entiende por precisión de un dato de salida? ¿A qué tipo de datos se aplica esto?

4.39. ¿Cómo se puede especificar la precisión en la función `printf`?

4.40. ¿Qué le ocurre a un número en coma flotante si se debe recortar para satisfacer la precisión especificada? ¿Y a una cadena de caracteres?

4.41. ¿Debe ir acompañada una especificación de precisión de una especificación de longitud de campo mínima en la función `printf`?

4.42. ¿Cómo se pueden indicar los argumentos de tipo entero corto, entero largo y de doble precisión en la cadena de control de la función `printf`? ¿Cómo se pueden indicar los argumentos `long double`?

4.43. ¿Cómo se interpretan los caracteres de conversión en mayúsculas en comparación con los caracteres en minúsculas correspondientes en la función `printf`? ¿A qué tipos de conversiones se aplica esto? ¿Permiten esta distinción todos los compiladores de C?

4.44. Mencionar el propósito de los indicadores comúnmente utilizados en la función `printf`.

4.45. ¿Pueden aparecer dos o más indicadores consecutivamente en el mismo grupo de caracteres?

4.46. ¿Cómo se interpretan los caracteres no reconocidos dentro de la cadena de control de una función `printf`?

4.47. ¿Cómo se pueden rotular los datos presentados por una función `printf`?

4.48. Mencionar el uso de las funciones `gets` y `puts` en la transferencia de cadenas de caracteres entre la computadora y los dispositivos de entrada/salida estándar. Comparar estas funciones con `scanf` y `printf`.

4.49. Explicar, en términos generales, cómo se puede generar un diálogo interactivo con el uso repetido de pares de funciones `scanf` y `printf`.

PROBLEMAS

4.50. Un programa en C contiene las siguientes instrucciones:

```
#include <stdio.h>

char a, b, c;
```

a) Escribir las instrucciones `getchar` oportunas que permitan introducir los valores de `a`, `b` y `c` en la computadora.
b) Escribir las instrucciones `putchar` que se ocupen de visualizar los valores presentes de `a`, `b` y `c`.

4.51. Resolver el Problema 4.50 utilizando sólo una función `scanf` y una `printf` en lugar de las instrucciones `getchar` y `putchar`. Comparar la respuesta con la solución al Problema 4.50.

4.52. Un programa en C contiene las siguientes instrucciones:

```
#include <stdio.h>

char texto[80];
```

a) Escribir una instrucción `for` que permita introducir un mensaje de 60 caracteres y se almacene en el array `texto`. Incluir una llamada a la función `getchar` en el bucle `for`, como en el Ejemplo 4.4.
b) Escribir una instrucción `for` que escriba los 60 primeros caracteres del array `texto`. Incluir una llamada a la función `putchar` en el bucle `for`, como en el Ejemplo 4.4.

4.53. Modificar la solución del Problema 4.52 *a*) para que se pueda introducir un array de caracteres de longitud no especificada. Supóngase que el mensaje no va a exceder nunca los 79 caracteres y que se termina con un carácter de *nueva línea* (`\n`). (Ver Ejemplo 4.4.)

4.54. Resolver el Problema 4.53 utilizando una instrucción `scanf` en lugar de la instrucción `for` (ver Ejemplo 4.8). ¿Qué información adicional se consigue con el método descrito en el Problema 4.53?

4.55. Un programa en C contiene las siguientes instrucciones:

```
#include <stdio.h>

int i, j, k;
```

Escribir de forma adecuada una función `scanf` que permita introducir los valores numéricos de `i`, `j` y `k`, asumiendo que:

a) Los valores de `i`, `j` y `k` son enteros decimales.
b) El valor de `i` es un entero decimal, `j` un entero octal y `k` un entero hexadecimal.
c) Los valores de `i` y `j` son enteros hexadecimales y `k` un entero octal.

4.56. Un programa en C contiene las siguientes instrucciones:

```
#include <stdio.h>

int i, j, k;
```

Escribir de forma adecuada una función `scanf` que permita introducir los valores numéricos de `i`, `j` y `k`, asumiendo que:

a) Los valores de `i`, `j` y `k` son enteros decimales y cada uno no excede los seis caracteres.
b) El valor de `i` es un entero decimal, `j` un entero octal y `k` un entero hexadecimal, y cada cantidad no excede los 8 caracteres.
c) Los valores de `i` y `j` son enteros hexadecimales y `k` un entero octal. Cada cantidad tiene 7 o menos caracteres.

4.57. Interpretar el significado de la cadena de control de cada función `scanf`:

a) `scanf("%12ld %5hd %15lf %15le", &a, &b, &c, &d);`
b) `scanf("%10lx %6ho %5hu %14lu", &a, &b, &c, &d);`
c) `scanf("%12D %hd %15f %15e", &a, &b, &c, &d);`
d) `scanf("%8d %*d %12lf %15lf", &a, &b, &c, &d);`

4.58. Un programa en C contiene las siguientes instrucciones:

```
#include <stdio.h>

int i, j;
long ix;
short s;
unsigned u;
float x;
double dx;
char c;
```

Escribir una función `scanf` para cada uno de los siguientes grupos de variables que lea un conjunto de datos correspondiente en la computadora y los asigne a las variables. Supóngase que todos los enteros se leerán como cantidades decimales.

a) `i, j, x y dx`
b) `i, ix, j, x y u`
c) `i, u y c`
d) `c, x, dx y s`

4.59. Un programa en C contiene las siguientes instrucciones:

```
#include <stdio.h>

int i, j;
long ix;
short s;
unsigned u;
float x;
double dx;
char c;
```

Escribir una función `scanf` que satisfaga cada uno de los siguientes puntos, suponiendo que todos los enteros se leerán como cantidades decimales.

a) Introducir los valores de `i`, `j`, `x` y `dx` suponiendo que cada cantidad entera no excede los cuatro caracteres, la cantidad en coma flotante no excede los ocho caracteres y la cantidad en doble precisión no excede los 15 caracteres.
b) Introducir los valores de `i`, `ix`, `j`, `x` y `u` suponiendo que cada cantidad entera no excede los cinco caracteres, el entero largo no tiene más de 12 caracteres y la cantidad en coma flotante no excede los 10 caracteres.
c) Introducir los valores de `i`, `u` y `c` suponiendo que cada cantidad entera no tiene más de seis caracteres.
d) Introducir los valores de `c`, `x`, `dx` y `s` suponiendo que la cantidad en coma flotante no excede los nueve caracteres, la cantidad en doble precisión no excede los 16 caracteres y el entero corto no excede los seis caracteres.

4.60. Un programa en C contiene las siguientes instrucciones:

```
#include <stdio.h>

char texto[80];
```

Escribir una función `scanf` que lea una cadena de caracteres y se le asigne al array de caracteres `texto`. Suponga que la cadena de caracteres no contiene ningún carácter de espaciado.

4.61. Resolver el Problema 4.60 suponiendo que la cadena de caracteres sólo contiene letras minúsculas, espacios en blanco y caracteres de nueva línea.

4.62. Resolver el Problema 4.60 suponiendo que la cadena de caracteres sólo contiene letras mayúsculas, dígitos, signos del dólar y espacios en blanco.

4.63. Resolver el Problema 4.60 suponiendo que la cadena de caracteres contiene cualquier carácter salvo el asterisco (supóngase que el asterisco se utiliza para indicar el final de la cadena).

4.64. Un programa en C contiene las siguientes instrucciones:

```
#include <stdio.h>

char a, b, c;
```

Supongamos que se desea introducir `$` y que se le asigne a `a`, que a `b` se le asigne `*` y que a `c` se le asigne `@`. Muestre cómo se deben introducir los datos para cada una de las siguientes funciones `scanf`:

a) `scanf("%c%c%c", &a, &b, &c);`
b) `scanf("%c %c %c", &a, &b, &c);`
c) `scanf("%s%s%s", &a, &b, &c);`
d) `scanf("%s %s %s", &a, &b, &c);`
e) `scanf("%1s%1s%1s", &a, &b, &c);`

4.65. Un programa en C contiene las siguientes instrucciones:

```
#include <stdio.h>

int a, b;
float x, y;
```

Supongamos que se desea introducir `12` y que le sea asignado a `a`, que `-8` le sea asignado a `b`, `0.011` le sea asignado a `x` y `-2.2x10`^{6} le sea asignado a `y`. Muestre cómo se deberían proporcionar los datos a la computadora al ejecutarse cada una de las siguientes funciones `scanf`:

a) `scanf("%d %d %f %f", &a, &b, &x, &y);`
b) `scanf("%d %d %e %e", &a, &b, &x, &y);`
c) `scanf("%2d %2d %5f %6e", &a, &b, &x, &y);`
d) `scanf("%3d %3d %8f %8e", &a, &b, &x, &y);`

4.66. Un programa en C contiene las siguientes instrucciones:

```
#include <stdio.h>

int i, j, k;
```

Escribir una función `printf` para cada uno de los siguientes grupos de variables o expresiones. Suponga que todas las variables representan enteros decimales.

a) `i, j y k`
b) `(i + j), (i - k)`
c) `sqrt(i + j), abs(i - k)`

4.67. Un programa en C contiene las siguientes instrucciones:

```
#include <stdio.h>

int i, j, k;
```

Escribir una función `printf` para cada uno de los siguientes grupos de variables o expresiones. Suponga que todas las variables representan enteros decimales.

a) `i, j` y `k`, con una longitud de campo mínima de tres caracteres por cantidad.
b) `(i + j), (i - k)`, con una longitud de campo mínima de cinco caracteres por cantidad.
c) `sqrt(i + j), abs(i - k)`, con una longitud de campo mínima de nueve caracteres para la primera cantidad y siete para la segunda.

4.68. Un programa en C contiene las siguientes instrucciones:

```
#include <stdio.h>

float x, y, z;
```

Escribir una función `printf` para cada uno de los siguientes grupos de variables o expresiones:

a) `x, y y z`
b) `(x + y), (x - z)`
c) `sqrt(x + y), fabs(x - z)`

4.69. Un programa en C contiene las siguientes instrucciones:

```
#include <stdio.h>

float x, y, z;
```

Escribir una función `printf` para cada uno de los siguientes grupos de variables o expresiones, utilizando la conversión tipo `f` para cada cantidad en coma flotante:

a) `x, y y z`, con una longitud de campo mínima de seis caracteres por cantidad.
b) `(x + y), (x - z)`, con una longitud de campo mínima de ocho caracteres por cantidad.
c) `sqrt(x + y), fabs(x - z)`, con una longitud de campo mínima de 12 caracteres para la primera cantidad y nueve para la segunda.

4.70. Repetir el problema anterior utilizando la conversión tipo `e`.

4.71. Un programa en C contiene las siguientes instrucciones:

```
#include <stdio.h>

float x, y, z;
```

Escribir una función `printf` para cada uno de los siguientes grupos de variables o expresiones, utilizando la conversión tipo `f` para cada cantidad en coma flotante:

a) `x, y y z`, con una longitud de campo mínima de ocho caracteres por cantidad, con sólo cuatro cifras decimales.
b) `(x + y), (x - z)`, con una longitud de campo mínima de nueve caracteres por cantidad, con tres cifras decimales.
c) `sqrt(x + y), fabs(x - z)`, con una longitud de campo mínima de 12 caracteres para la primera cantidad y 10 para la segunda. Presentar un máximo de cuatro cifras decimales en cada cantidad.

4.72. Un programa en C contiene las siguientes instrucciones:

```
#include <stdio.h>

float x, y, z;
```

Escribir una función `printf` para cada uno de los siguientes grupos de variables o expresiones, utilizando la conversión tipo `e` para cada cantidad en coma flotante:

a) `x, y y z`, con una longitud de campo mínima de 12 caracteres por cantidad, con cuatro cifras decimales.
b) `(x + y), (x - z)`, con una longitud de campo mínima de 14 caracteres por cantidad, con cinco cifras decimales.
c) `sqrt(x + y), fabs(x - z)`, con una longitud de campo mínima de 12 caracteres para la primera cantidad y 15 para la segunda. Presentar un máximo de siete cifras decimales en cada cantidad.

4.73. Un programa en C contiene las siguientes instrucciones:

```
#include <stdio.h>

int a = 0177, b = 055, c = 0xa8, d = 0x1ff;
```

Escribir una función `printf` para cada uno de los siguientes grupos de variables o expresiones:

a) `a, b, c y d`
b) `(a + b), (c - d)`

4.74. Un programa en C contiene las siguientes instrucciones:

```
#include <stdio.h>

int i, j;
long ix;
unsigned u;
float x;
double dx;
char c;
```

Escribir una función `printf` para cada uno de los siguientes grupos de variables. Suponga que todos los enteros se desean presentar como cantidades decimales.

a) `i, j, x y dx`
b) `i, ix, j, x y u`
c) `i, u y c`
d) `c, x, dx e ix`

4.75. Un programa en C contiene las siguientes instrucciones:

```
#include <stdio.h>

int i, j;
long ix;
unsigned u;
float x;
double dx;
char c;
```

Escribir una función `printf` para cada una de las siguientes situaciones, suponiendo que todos los enteros se desean presentar como cantidades decimales.

a) Escribir los valores de `i`, `j`, `x` y `dx` suponiendo que cada cantidad entera tiene una longitud de campo mínima de cuatro caracteres y cada cantidad en coma flotante se presenta en notación exponencial con un total de al menos 14 caracteres y no más de ocho cifras decimales.
b) Repetir *a*), visualizando cada cantidad en una línea.
c) Escribir los valores de `i`, `ix`, `j`, `x` y `u` suponiendo que cada cantidad entera tendrá una longitud de campo mínima de cinco caracteres, el entero largo tendrá una longitud de campo mínima de 12 caracteres y la cantidad en coma flotante tiene al menos 10 caracteres con un máximo de cinco cifras decimales. No incluir el exponente.
d) Repetir *c*), visualizando las tres primeras cantidades en una línea, seguidas de una línea en blanco y las otras dos cantidades en la línea siguiente.
e) Escribir los valores de `i`, `u` y `c`, con una longitud de campo mínima de seis caracteres para cada cantidad entera. Separar con tres espacios en blanco cada cantidad.
f) Escribir los valores de `j`, `u` y `x`. Visualizar las tres cantidades enteras con una longitud de campo mínima de cinco caracteres. Presentar la cantidad en coma flotante utilizando la conversión tipo `f`, con una longitud de campo mínima de 11 caracteres y un máximo de cuatro cifras decimales.
g) Repetir *f*), con cada dato ajustado a la izquierda dentro de su campo.
h) Repetir *f*), apareciendo un signo (tanto + como -) delante de cada dato con signo.
i) Repetir *f*), rellenando el campo de cada entidad entera con ceros.
j) Repetir *f*), apareciendo el valor de `x` con un punto decimal.

4.76. Supongamos que `i`, `j` y `k` son variables enteras y que `i` representa una cantidad octal, `j` una cantidad decimal y `k` una cantidad hexadecimal. Escribir una función `printf` adecuada para cada una de las siguientes situaciones.

a) Escribir el valor de `i`, `j` y `k`, con una longitud de campo mínima de ocho caracteres por cada valor.
b) Repetir *a*), con cada dato ajustado a la izquierda de su campo.
c) Repetir *a*), con cada dato precedido de ceros (`0x`, en el caso de la cantidad hexadecimal).

4.77. Un programa en C contiene las siguientes declaraciones de variables:

```
int i = 12345, j = -13579, k = -24680;
long ix = 123456789;
short sx = -2222;
unsigned ux = 5555;
```

Mostrar la salida resultante de cada una de las siguientes instrucciones `printf`:

a) `printf("%d %d %d %ld %d %u", i, j, k, ix, sx, ux);`
b) `printf("%3d %3d %3d\n\n%3ld %3d %3u", i, j, k, ix, sx, ux);`
c) `printf("%8d %8d %8d\n\n%15ld %8d %8u", i, j, k, ix, sx, ux);`
d) `printf("%-8d %-8d\n%-8d %-15ld\n%-8d %-8u", i, j, k, ix, sx, ux);`
e) `printf("%+8d %+8d\n%+8d %+15ld\n%+8d %8u", i, j, k, ix, sx, ux);`
f) `printf("%08d %08d\n%08d %015ld\n%08d %08u", i, j, k, ix, sx, ux);`

4.78. Un programa en C contiene las siguientes declaraciones de variables:

```
int i = 12345, j = 0xabcd9, k = 077777;
```

Mostrar la salida resultante de cada una de las siguientes instrucciones `printf`:

a) `printf("%d %x %o", i, j, k);`
b) `printf("%3d %3x %3o", i, j, k);`
c) `printf("%8d %8x %8o", i, j, k);`
d) `printf("%-8d %-8x %-8o", i, j, k);`
e) `printf("%+8d %+8x %+8o", i, j, k);`
f) `printf("%08d %#8x %#8o", i, j, k);`

4.79. Un programa en C contiene las siguientes declaraciones de variables:

```
float a = 2.5, b = 0.0005, c = 3000.;
```

Mostrar la salida resultante de cada una de las siguientes instrucciones `printf`:

a) `printf("%f %f %f", a, b, c);`
b) `printf("%3f %3f %3f", a, b, c);`
c) `printf("%8f %8f %8f", a, b, c);`
d) `printf("%8.4f %8.4f %8.4f", a, b, c);`
e) `printf("%8.3f %8.3f %8.3f", a, b, c);`
f) `printf("%e %e %e", a, b, c);`
g) `printf("%3e %3e %3e", a, b, c);`
h) `printf("%12e %12e %12e", a, b, c);`
i) `printf("%12.4e %12.4e %12.4e", a, b, c);`
j) `printf("%8.2e %8.2e %8.2e", a, b, c);`
k) `printf("%-8f %-8f %-8f", a, b, c);`
l) `printf("%+8f %+8f %+8f", a, b, c);`
m) `printf("%08f %08f %08f", a, b, c);`
n) `printf("%#8f %#8f %#8f", a, b, c);`
o) `printf("%g %g %g", a, b, c);`
p) `printf("%#g %#g %#g", a, b, c);`

4.80. Un programa en C contiene las siguientes declaraciones de variables:

```
char c1 = 'A', c2 = 'B', c3 = 'C';
```

Mostrar la salida resultante de cada una de las siguientes instrucciones `printf`:

a) `printf("%c %c %c", c1, c2, c3);`
b) `printf("%c%c%c", c1, c2, c3);`

c) `printf("%3c %3c %3c", c1, c2, c3);`
d) `printf("%3c%3c%3c", c1, c2, c3);`
e) `printf("c1=%c c2=%c c3=%c", c1, c2, c3);`

4.81. Un programa en C contiene las siguientes instrucciones:

```
#include <stdio.h>

char texto[80];
```

Escribir una función `printf` para visualizar el contenido de `texto` de las siguientes formas:

a) Todo en una línea.
b) Sólo los ocho primeros caracteres.
c) Los ocho primeros caracteres, precedidos de cinco espacios en blanco.
d) Los ocho primeros caracteres, seguidos de cinco espacios en blanco.

4.82. Un programa en C contiene la siguiente declaración de array:

```
char texto[80];
```

Supongamos que se le ha asignado la siguiente cadena de caracteres a `texto`:

```
Programar en C puede ser una actividad enormemente creativa.
```

Mostrar la salida resultante de cada una de las siguientes instrucciones `printf`:

a) `printf("%s", texto)`
b) `printf("%18s", texto);`
c) `printf("%.18s", texto);`
d) `printf("%18.7s", texto);`
e) `printf("%-18.7s", texto);`

4.83. Escribir las instrucciones `scanf` o `printf` necesarias para cada uno de los siguientes puntos:

a) Generar el mensaje:

```
Por favor, introduce tu nombre:
```

y que el usuario introduzca en la misma línea su nombre. Asignar el nombre a un array de caracteres llamado `nombre`.

b) Supongamos que `x1` y `x2` son variables en coma flotante cuyos valores son `8.0` y `-2.5`, respectivamente. Visualizar los valores de `x1` y `x2`, con los rótulos adecuados, por ejemplo:

```
x1 = 8.0     x2 = -2.5
```

c) Supongamos que `a` y `b` son variables enteras. Pedir al usuario que introduzca el valor de estas dos variables y mostrar después su suma. Rotular la salida adecuadamente.

4.84. Determinar de qué caracteres de conversión dispone en su versión de C. Determinar también de qué indicadores se dispone para la salida de datos.

CAPÍTULO 5

Preparación y ejecución de un programa en C

Con lo visto hasta el momento hemos aprendido lo suficiente para escribir programas completos en C, aunque sencillos. Vamos a detener brevemente la presentación de nuevos conceptos y vamos a prestar alguna atención a la planificación, escritura y ejecución de un programa en C completo. Además, trataremos algunos métodos para la detección y corrección de los diferentes tipos de errores que pueden surgir en programas mal escritos.

Dirigiremos nuestra atención hacia el uso de Turbo C++ versión 4.5 de Borland International, ejecutándose bajo el entorno operativo Windows (recordar que C++ incluye una implementación completa del estándar ANSI C, como vimos en la sección 1.5). Se hace especial énfasis en esta versión concreta de C debido a la gran popularidad de las computadoras personales, su bajo cos-te y también porque es representativo de la utilización de C en muchas otras computadoras.

5.1. PLANIFICACIÓN DE UN PROGRAMA EN C

Es esencial que se encuentre trazada completamente la estrategia del programa en conjunto antes de comenzar realmente con los detalles de la programación. Esto permite concentrarnos en la lógica del programa en general, sin perdernos en los detalles sintácticos de las instrucciones reales. Una vez que se ha fijado claramente la estrategia del programa en conjunto, podemos pasar a considerar los detalles asociados a las instrucciones individuales del programa. Este enfoque se conoce normalmente como programación «descendente» («top-down»). Este proceso completo se puede repetir varias veces en programas grandes, añadiendo en cada estado más detalles de programación.

La organización descendente del programa se efectúa al desarrollar un esquema informal que consta de frases o sentencias compuestas parte en castellano y parte en C. En los estados iniciales del desarrollo del programa la cantidad de C es mínima, utilizando algunos elementos para definir los componentes principales del programa, tales como las cabeceras de las funciones, llamadas a funciones, llaves que determinan instrucciones compuestas y partes de sentencias de control que describen las estructuras principales del programa. Se introduce entre estos elementos material descriptivo en castellano, frecuentemente en forma de comentarios. El esquema resultante se suele denominar *pseudocódigo*.

EJEMPLO 5.1. **Interés compuesto.** Un problema que se presenta frecuentemente en las finanzas domésticas es el determinar cuánto dinero se acumulará en una cuenta en el banco después de *n* años si se

deposita inicialmente una cantidad conocida P, y la cuenta acumula anualmente interés a un tanto por ciento anual r. La respuesta a esta cuestión se puede determinar mediante la bien conocida fórmula

$$F = P\,(1 + i)^n$$

en donde F representa la cantidad futura de dinero (incluyendo la suma original P, denominada el *principal*) e i es la representación decimal del tanto por ciento de interés; esto es, i=r/100 (por ejemplo, un interés de r=5% se corresponde con i=0.05).

Veamos cómo organizar un programa en C que resuelva este problema. El programa se basará en el siguiente esquema:

1. Declarar las variables del programa requeridas.
2. Leer los valores del principal (P), el tanto por ciento de interés (r) y el número de años (n).
3. Calcular la representación decimal del interés utilizando la fórmula

   $$i = r/100$$

4. Determinar la cantidad futura (F) utilizando la fórmula

   $$F = P\,(1 + i)^n$$

5. Escribir el valor calculado de F.

   Veamos a continuación el pseudocódigo correspondiente al esquema anterior.

```
/* cálculo del interés compuesto */

main()

{
    /* declarar las variables del programa */
    /* leer los valores de P, r y n */
    /* calcular el valor de i */
    /* calcular el valor de F */
    /* escribir el valor calculado de F */
}
```

En principio todos estos pasos parecen muy sencillos. Sin embargo, alguno de ellos requiere más detalle antes de que se pueda programar realmente. Por ejemplo, la entrada de datos se efectuará de forma interactiva. Esto requerirá algún tipo de diálogo que se podrá generar mediante pares de sentencias printf y scanf, como se explicó en el Capítulo 4. Además, C no tiene operador de potenciación. Por tanto, se necesita algún detalle adicional para evaluar la fórmula

$$F = P\,(1 + i)^n$$

He aquí una versión más detallada del anterior esquema.

```
/* cálculo del interés compuesto */

main()
{
    /* declarar las variables de coma flotante p, r, n, i y f */
```

```
    /* pedir el valor de p y leer su valor */
    /* pedir el valor de r y leer su valor */
    /* pedir el valor de n y leer su valor */

    /* calcular i = r/100 */

    /* calcular f = p(1+i)ⁿ como sigue:

       f = p * pow((1+i),n)

       en donde pow es una función de biblioteca para la potenciación */

    /* escribir el valor de f con el correspondiente rótulo */
}
```

Este esquema puede incluir más detalles de los realmente necesarios para un programa tan sencillo, pero sirve como ejemplo del enfoque descendente en el desarrollo de programas.

En los Ejemplos 5.2, 5.4 y 5.5 veremos el desarrollo detallado y la implementación de este programa.

Otro método que a veces se utiliza al planificar un programa en C es el enfoque ascendente («bottom-up»). Este método puede ser útil para programas que hacen un uso importante de módulos (por ejemplo funciones definidas por el usuario). El enfoque ascendente conlleva el desarrollo detallado de estos módulos al comienzo del proceso de planificación. El desarrollo de todo el programa se basa, por tanto, en las características conocidas de estos módulos de los que se dispone.

En la práctica de suelen utilizar los dos enfoques: el descendente para la planificación del programa en conjunto, la ascendente para el desarrollo de módulos antes de la parte principal del programa y la descendente con respecto al desarrollo de cada módulo individualmente.

5.2. ESCRITURA DE UN PROGRAMA EN C

Una vez determinada la estrategia general del programa y escrito un esquema del mismo, la atención pasa al desarrollo detallado de un programa en C que funcione. En este momento hemos de conseguir traducir cada paso del esquema del programa (o cada porción del pseudocódigo) a una o más instrucciones equivalentes en C. Esto debería ser una tarea relativamente sencilla una vez fijada la estructura general del programa con el suficiente detalle.

La escritura de un programa completo en C es algo más que disponer las declaraciones e instrucciones adecuadas en el orden correcto con los signos de puntuación adecuados. Se debe prestar atención a la inclusión de ciertos elementos que aumenten la legibilidad del programa y la salida resultante. En estos elementos se incluye el secuenciamiento lógico de las sentencias, el sangrado adecuado de éstas, la inclusión de comentarios y la generación de salida rotulada de forma apropiada.

La elección de las instrucciones del programa y su secuenciamiento lógico dentro del mismo está determinada en gran manera por la lógica subyacente del programa. Existirán a menudo varias formas de conseguir el mismo resultado. Esto es cierto en mayor medida en programas

más complicados que involucran el uso de segmentos de forma repetida o condicional. En estos casos la forma en que está organizado el programa puede tener un mayor efecto en la claridad de éste y en su eficiencia al ser ejecutado. Por tanto, es importante que se seleccionen las instrucciones y su secuenciamiento de la manera más efectiva. Hablaremos más de esto en al Capítulo 6, en donde discutimos los elementos disponibles en C para la ejecución condicional y repetida.

El sangrado de las instrucciones está íntimamente relacionado con el secuenciamiento de los grupos de éstas dentro del programa. Mientras que el secuenciamiento afecta al orden en que efectúan un grupo de operaciones, el sangrado indica la forma en que se encuentran subordinadas las instrucciones dentro de un grupo. Además, a veces se utilizan líneas en blanco para separar grupos de instrucciones relacionados. Las ventajas del sangrado y de las líneas en blanco resultan obvias, incluso en los programas sencillos presentados anteriormente en este libro. Lo serán aún más a la vez que veamos programas en C con estructuras más complicadas.

Siempre se deben incluir comentarios en un programa en C. Escritos con propiedad, pueden proporcionar una visión general de la lógica del programa muy útil. También pueden ayudar a determinar las partes principales del programa, identificar ciertos elementos clave y proporcionar otra información útil. Generalmente, los comentarios no necesitan ser extensos; se puede conseguir, mediante unos comentarios breves y bien situados, hacer claro un programa que sin ellos no lo sería. Estos comentarios pueden ser muy útiles para cualquiera que intente leer y entender el programa y para el mismo programador, ya que la mayoría de los programadores suele olvidar los detalles de sus propios programas después de un cierto tiempo. Esto es especialmente cierto en programas largos y complicados.

Otra característica importante de los programas bien escritos es la generación de una salida clara y legible. Existen dos factores que contribuyen a esta legibilidad. El primero es rotular los datos de salida, como discutimos en el Capítulo 4. El segundo es la aparición en la salida de algunos de los datos de entrada, de forma que se pueda identificar cada instanciación (si hay más de una) en la ejecución del programa. La forma en la que esto se haga depende del entorno en el que se vaya a ejecutar el programa en C. En un entorno interactivo los datos de entrada se visualizarán normalmente en el terminal al mismo tiempo que se introduzcan durante la ejecución del programa. Por tanto, no será necesario que los datos de entrada se escriban otra vez.

Cuando se ejecuta un programa interactivo, el *usuario* (alguien distinto del programador) puede no saber cómo introducir los datos requeridos. Por ejemplo, el usuario puede no saber qué datos introducir, cuándo introducirlos o el orden en lo que debe hacer. Por consiguiente, un programa interactivo bien escrito debe generar *mensajes* y *rótulos* en los momentos adecuados durante la ejecución del programa para proporcionar esta información.

EJEMPLO 5.2. **Interés compuesto.** Consideremos un programa interactivo correspondiente al esquema presentado en el Ejemplo 5.1.

```
/* problema sencillo de interés compuesto */

#include <stdio.h>
#include <math.h>

main()

{
    float p, r, n, i, f;
```

```
    /* leer datos de entrada (mediante peticiones rotuladas) */

    printf("Por favor, introduce la suma inicial (P): ");

    scanf("%f", &p);
    printf("Por favor, introduce el interés (r): ");
    scanf("%f", &r);
    printf("Por favor, introduce el número de años (n): ");
    scanf("%f", &n);

    /* calcular i y f */

    i = r/100;
    f = p * pow((1 + i), n);

    /* escribir la salida */

    printf("\nEl valor final (F) es : %.2f\n", f);
}
```

El programa de este ejemplo es lógicamente muy simple. No hemos tenido, pues, que preocuparnos de considerar otras formas alternativas al secuenciamiento de las instrucciones. Sin embargo, podríamos haber incluido otros elementos deseables. Por ejemplo, podríamos haber deseado que el programa se ejecutase repetidamente para diferentes conjuntos de datos de entrada. O podríamos haber previsto diferentes errores, advirtiendo al usuario de la incorrección de los parámetros de entrada. En el Capítulo 6 veremos cómo se pueden añadir estas mejoras.

5.3. INTRODUCCIÓN DE UN PROGRAMA EN LA COMPUTADORA

Una vez que se ha escrito el programa, se debe introducir en la computadora antes de que se pueda compilar y ejecutar. En las versiones antiguas de C esto se realizaba escribiendo para el programa un archivo de texto línea a línea, mediante un editor o un procesador de textos.

La mayoría de las versiones de C o C++ actuales incluyen para este propósito un *editor de pantalla*. El editor se encuentra integrado normalmente en el entorno software. De esta forma, para acceder al editor se debe entrar primero en el entorno de programación C o C++. La forma de hacer esto cambia de una implementación de C a otra.

Consideremos, por ejemplo, la versión 4.5 de Turbo C++, ejecutándose bajo Windows en una computadora personal, compatible IBM. Para entrar en Turbo C++, se restaura el grupo Turbo C++ y a continuación se pulsa el icono Turbo C++. Aparecerá una ventana casi vacía como la mostrada en la Figura 5.1. La primera línea de esta ventana (la cual contiene `Turbo C++ -[noname00.cpp]`) es la *barra del título*, y la segunda línea (que contiene `File Edit Search View`, etc.) es la *barra del menú*. La selección de un elemento de la barra del menú hará aparecer un *menú desplegable*, con una serie de opciones relativas a la selección de la barra del menú. Por ejemplo, el menú `File` incluye opciones que permiten abrir un nuevo programa, recuperar un programa existente, guardar un programa, imprimir el listado de un programa, o salir de Turbo C++. Trataremos algunos de estos menús desplegables más adelante en este capítulo.

Normalmente se utiliza un dispositivo apuntador, tal como un *ratón*, para seleccionar un elemento de un menú. Esto se hace situando el cursor sobre el elemento deseado y luego «haciendo clic» sobre el elemento; es decir, pulsando un botón del dispositivo apuntador.

El gran espacio en blanco por debajo de la barra del menú constituye el *área de edición* donde se puede introducir un nuevo programa o visualizar un programa existente. Partes del programa listado en este área se pueden modificar, borrar, copiar o desplazar a otra parte del programa. Algunos de estos cambios se realizan directamente en el área de edición, mientras que otros se realizan *destacando* (es decir, seleccionando) una parte del programa y luego copiando, desplazando o borrando el material destacado utilizando las opciones del menú `Edit`. La selección se efectúa normalmente manteniendo pulsado un botón del ratón y luego arrastrando el ratón a través del material a seleccionar.

Las *barras de desplazamiento* se encuentran debajo y a la derecha del área de edición. Las barras de desplazamiento permiten movernos rápidamente a otras partes del programa cuando su listado se extiende más allá de los límites de la pantalla. De esta forma, podemos desplazarnos verticalmente a través del listado del programa pulsando a lo largo de la barra de desplazamiento de la derecha, o desplazando el cuadro de desplazamiento arriba o abajo. De forma análoga, podemos desplazarnos horizontalmente a través del listado del programa pulsando a lo largo de la barra de desplazamiento de la parte inferior, o arrastrando el cuadro de desplazamiento a la derecha o a la izquierda.

Finalmente, la última línea es la *barra de estado*, que indica el estado actual del área de edición o el propósito de la opción del menú seleccionada actualmente. En la Figura 5.1 se indica que la ventana de edición está en la modalidad *insertar*, lo que significa que el texto puede insertarse en cualquier parte dentro de la ventana.

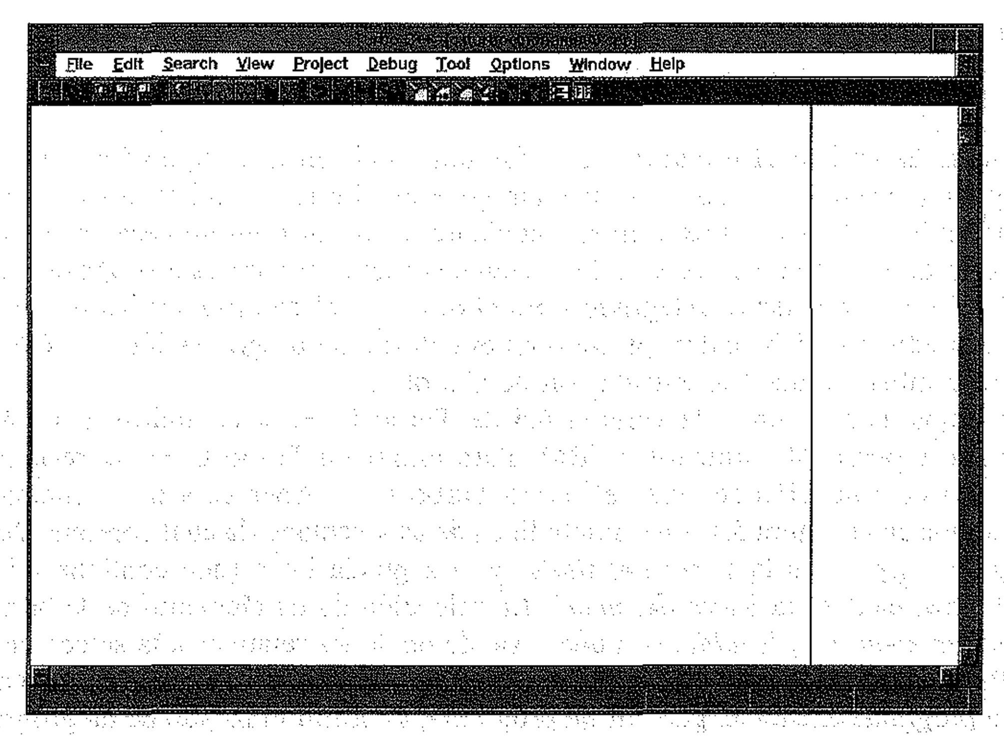


Figura 5.1.

Para introducir un programa nuevo en Turbo C++, sencillamente se teclea el programa en el área de edición línea a línea y se pulsa la tecla `Intro` al final de cada línea. Para modificar una línea, se utiliza el ratón o las teclas de movimiento del cursor (teclas de flecha) para situar el cursor al comienzo del área a editar. A continuación se utilizan las teclas `Retroceso` y `Suprimir` para borrar los caracteres no deseados. También se pueden insertar caracteres adicionales, si es necesario.

Se pueden *borrar* una o más líneas simplemente destacándolas y luego seleccionando `Cut` del menú `Edit`, o pulsando la tecla `Suprimir`. Se puede *desplazar* un bloque de líneas de una posición a otra usando las opciones `Cut` y `Paste` del menú `Edit`. De forma análoga, se puede *copiar* un bloque de líneas de una posición a otra usando las opciones `Copy` y `Paste` del menú `Edit`. En el Manual del Usuario de Turbo C++ encontrará instrucciones de edición adicionales.

Una vez introducido el programa, antes de ejecutarlo es conveniente guardarlo en disco. Esto se realiza en Turbo C++ seleccionando la opción `Save As` del menú `File`, suministrando a continuación un nombre de programa, por ejemplo `INTEREST.C` (la extensión `C` se añadirá automáticamente si no se incluye ninguna extensión como parte del nombre del archivo.) Una vez que el programa ha sido guardado y se le ha dado un nombre, puede ser guardado de nuevo más adelante (por ejemplo, con los últimos cambios realizados), simplemente seleccionando `Save` del menú `File`.

Un programa que ha sido guardado puede ser reeditado posteriormente seleccionando `Open` en el menú `File`, y luego tecleando el nombre del programa o seleccionando el nombre de una lista de programas almacenados. En cualquier momento se puede obtener un listado impreso del programa actual (una copia en papel) seleccionando `Print` del menú `File`.

EJEMPLO 5.3. Interés compuesto. Supongamos que hemos introducido el programa del interés compuesto mostrado en el Ejemplo 5.2 en una computadora personal compatible IBM, utilizando Turbo C++. Tras realizar las correcciones pertinentes, la pantalla será similar a la mostrada en la Figura 5.2. A continuación se puede guardar el programa seleccionando `Save As` del menú `File`, como se muestra en la Figura 5.3. (Ambas figuras en página siguiente.)

Al seleccionar `Save As` aparecerá un cuadro de diálogo solicitando el nombre del programa a guardar. Respondamos introduciendo el nombre del programa `INTEREST.C`. A continuación, se puede concluir la sesión seleccionando `Exit` del menú `File`.

5.4. COMPILACIÓN Y EJECUCIÓN DEL PROGRAMA

Una vez que el programa ha sido introducido, corregido y guardado, se puede compilar y ejecutar seleccionando `Run` del menú `Debug`. Se abre una nueva ventana y se realiza un intento de compilar el programa actual. Si el programa no se compila correctamente, aparecerá en una ventana diferente un listado de mensajes de error. Cada mensaje de error indica el número de la línea donde se detectó el error, así como el tipo de error. Sin embargo, si el programa se compila con éxito, se empezará a ejecutar inmediatamente en una nueva ventana, pidiendo los datos de entrada, presentando la salida, etc.

EJEMPLO 5.4. Interés compuesto. Supongamos que volvemos a entrar en Turbo C++ tras concluir la sesión descrita en el Ejemplo 5.3. Empezamos cargando el programa anterior, `INTEREST.C`, en la memoria de la computadora, seleccionando `Open` en el menú `File`. A continuación seleccionamos `Run` en el menú `Debug`, como se muestra en la Figura 5.4.

File Edit Search View Project Debug Tool Options Window Help

```
/* Problema sencillo de interés compuesto */

#include <stdio.h>
#include <math.h>

main()

{

      float p, r, n, i, f;

      /* leer data de entrada mediante peticiones rotuladas */

      printf("Por favor introduce la suma inicial (P): ");
      scanf("%f", &p);
      printf(Por favor, introduce el interés (r): ");
      scanf("%f", &r);
      printf(Por favor, introduce el número de años (n): ");
      scanf("%f", &n);

      /* calcular i y f */

      i = r / 100;
      f = p * pow((1 + i), n);

      /* escribir la salida */

      printf("\nEl valor final (F) es: %.2f\n", f);
}
```

Figura 5.2.

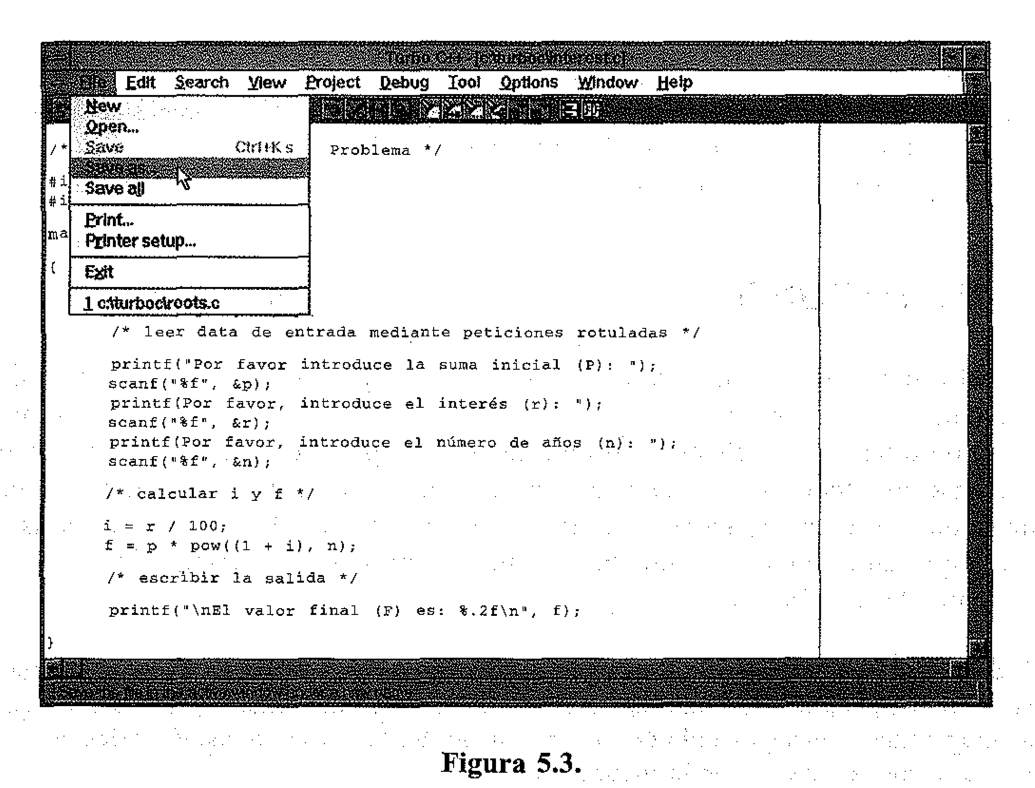


Figura 5.3.

```
File  Edit  Search  View  Project        Tool  Options  Window  Help

                                  Step over             F8
/* Problema sencillo de inte      Trace into            F7

#include <stdio.h>                Toggle breakpoint     F5
#include <math.h>                 Find execution point
                                  Pause program
main()                            Terminate program     Ctrl+F2

{                                 Add watch...          Ctrl+F5
     float p, r, n, i, f;         Add breakpoint...

     /* leer data de entrada      Evaluate/Modify...    Ctrl+F7   adas */
                                  Inspect...
     printf("Por favor introd                                       );
     scanf("%f", &p);             Load symbol table...
     printf(Por favor, introduce el interés (r): ");
     scanf("%f", &r);
     printf(Por favor, introduce el número de años (n): ");
     scanf("%f", &n);

     /* calcular i y f */

     i = r / 100;
     f = p * pow((1 + i), n);

     /* escribir la salida */

     printf("\nEl valor final (F) es: %.2f\n", f);
}
```

Figura 5.4.

El programa se compila con éxito e inmediatamente empieza a ejecutarse. Sobre la ventana original que contiene el listado del programa, aparece una nueva ventana que muestra el diálogo de entrada/salida. Ésta es mostrada en la Figura 5.5 para los valores de `P=1000`, `r=6` y `n=20`. Estos valores han sido introducidos por el usuario en respuesta a las peticiones de entrada.

Una vez introducida la última cantidad `(n=20)`, el programa reanuda su ejecución, generando la salida final mostrada en la Figura 5.6. De esta forma, vemos que para las cantidades de entrada dadas, se obtiene un valor de `F=3207.14`.

5.5. MENSAJES DE ERROR

Los errores de programación suelen permanecer ocultos hasta que se hace un intento de compilación o ejecución del programa. La presencia de errores *sintácticos* (o *gramaticales*) se hará patente rápidamente tras ejecutar la orden `Run`, ya que estos errores impiden que el programa sea compilado o ejecutado con éxito. Algunos errores particularmente frecuentes de este tipo son variables inapropiadamente declaradas, referenciar una variable no declarada, puntuación incorrecta, etc.

La mayoría de los compiladores de C generarán *mensajes de error* cuando se detecten errores sintácticos durante el proceso de compilación. Estos mensajes de error no siempre son claros en su significado y puede que no identifiquen correctamente la localización del error (aunque intentan hacerlo). A pesar de ello, suelen servir de gran ayuda en la identificación y localización aproximada de los errores.

Si un programa contiene varios errores sintácticos, puede que no se detecten todos en la primera pasada del compilador. Puede, por tanto, ser necesario corregir algunos errores sintácticos antes de que se puedan localizar otros. Este proceso puede repetirse varias veces antes de identificar y corregir todos los errores sintácticos.

Turbo C++ - [c:\turboc\interest.c]

File Edit Search View Project Debug Tool Options Window Help

```
/* Problema sencillo de interés compuesto */

#include <stdio.h>
#include <math.h>

main()

{
      float p, r
      /* leer da
      printf("Po
      scanf("%f"
      printf(Por
      scanf("%f"
      printf(Por
      scanf("%f"
      /* calcula

      i = r / 100;
      f = p * pow((1 + i), n);

      /* escribir la salida */

      printf("\nEl valor final (F) es: %.2f\n", f);
}
```

```
Por favor, introduce la suma inicial (P): 1000
Por favor, introduce el interés (r): 6
Por favor, introduce el número de años (n): 20
```

Figura 5.5.

Turbo C++ - [c:\turboc\interest.c]

File Edit Search View Project Debug Tool Options Window Help

```
/* Problema sencillo de interés compuesto */

#include <stdio.h>
#include <math.h>

main()

{
      float p, r
      /* leer da
      printf("Po
      scanf("%f"
      printf(Por
      scanf("%f"
      printf(Por
      scanf("%f"
      /* calcula

      i = r / 100;
      f = p * pow((1 + i), n);

      /* escribir la salida */

      printf("\nEl valor final (F) es: %.2f\n", f);
}
```

```
Por favor, introduce la suma inicial (P): 1000
Por favor, introduce el interés (r): 6
Por favor, introduce el número de años (n): 20

El valor final (F) es: 3207.14
```

Figura 5.6.

EJEMPLO 5.5. Errores sintácticos. He aquí otra versión del programa de interés compuesto presentado en los Ejemplos 5.2 a 5.4.

```
/* problema sencillo de interés compuesto */

#include <stdio.h>
include <math.h>

main()
{
   float p, r, n, i, f;

   /* leer datos de entrada (mediante peticiones rotuladas) */

   printf("Por favor, introduce la suma inicial (P): ");
   scanf("%f", &p);
   printf("Por favor, introduce el interés (r): );
   scanf("%f", &r);
   printf("Por favor, introduce el número de años (n): ");
   scanf("%f", &n)

   /* calcular i y f */

   i = r/100;
   f = p * pow((1 + i), n);

   /* escribir la salida /*

   printf("\nEl valor final (F) es : %.2f\n", f);
}
```

Esta versión del programa contiene cinco errores sintácticos diferentes. Son los siguientes:

1. La segunda instrucción `include` no comienza con un signo #.
2. La cadena de control de la segunda instrucción `printf` no tiene las comillas de cierre.
3. La última instrucción `scanf` no finaliza con un punto y coma.
4. La instrucción de asignación de `f` no tiene los paréntesis emparejados.
5. El último comentario finaliza de forma incorrecta (termina con `/*` en lugar de `*/`).

Cuando se intenta compilar el programa (bien seleccionando la opción `Run` del menú `Debug` o bien seleccionando `Compile` del menú `Project`), se obtienen en una ventana de mensajes aparte los mensajes de error mostrados en la Figura 5.7.

El primer mensaje hace referencia al signo # que falta en la línea 4 (la numeración de las líneas incluye las líneas vacías). El segundo mensaje denota la falta de unas comillas (`"`) al final de la segunda sentencia `printf` (línea 15), y el tercer mensaje indica la terminación inapropiada del último comentario (línea 25). Observe que los mensajes de error son muy crípticos (además de estar en inglés). Es por tanto necesario algo de ingenio para determinar lo que significan.

Tras identificar y corregir correctamente estos tres errores, se intenta otra vez compilar el programa. El resultado es el nuevo conjunto de mensajes de error mostrado en la Figura 5.8.

El primer mensaje de error indica que falta un punto y coma al final de la última instrucción `scanf` (lo que realmente ocurre en la línea 18 y no en la 22). El segundo mensaje indica que falta el paréntesis izquierdo

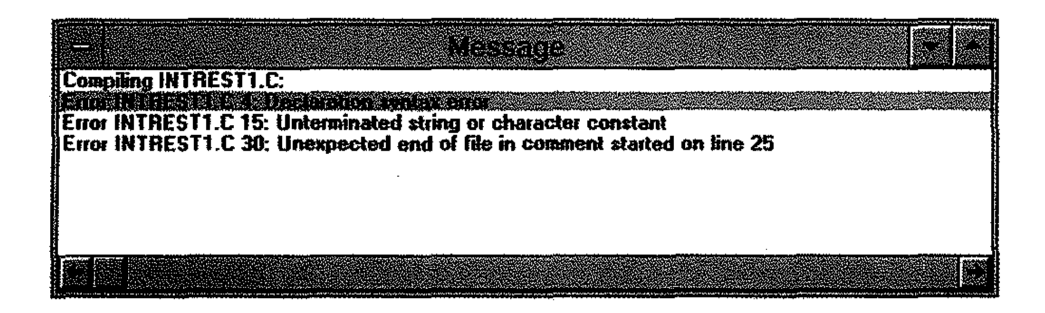


Figura 5.7.

en la segunda instrucción de asignación (línea 23). Las dos advertencias siguientes y el tercer mensaje de error son consecuencia de este mismo error.

Cuando se corrigen estos dos errores, el programa se compila correctamente y comienza a ejecutarse, como se muestra en la Figura 5.5.

Hay que tener presente que los mensajes de error y las advertencias varían de una versión de C a otra. Algunos compiladores pueden generar mensajes más largos o más informativos que los mostrados en este ejemplo, aunque los mostrados aquí son muy típicos.

Otro tipo común de errores es el error de *ejecución*. Los errores de ejecución ocurren durante la ejecución del programa, después de una compilación efectuada con éxito. Por ejemplo, algunos errores de ejecución comunes son el *«overflow»* o *«underflow» numérico* (exceder el mayor o menor número permitido que se puede almacenar en la computadora), división por cero, intentar calcular el logaritmo o la raíz cuadrada de un número negativo, etc. Normalmente en estas situaciones se generarán mensajes de error que harán fácil la detección y corrección de los mismos. Estos mensajes se suelen llamar de *ejecución* para distinguirlos de los de *compilación* descritos anteriormente.

EJEMPLO 5.6. Raíces reales de una ecuación cuadrática. Supongamos que queremos calcular las raíces reales de la ecuación cuadrática

$$ax^2 + bx + c = 0$$

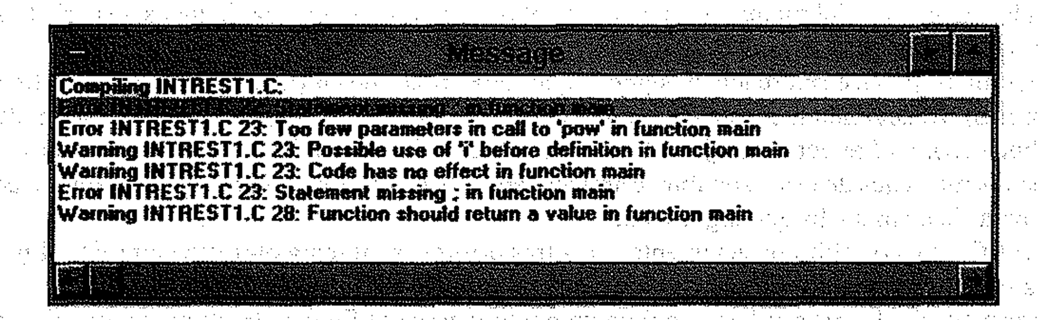


Figura 5.8.

utilizando la fórmula

$$x = \frac{-b \pm (b^2 - 4ac)^{1/2}}{2a}$$

Presentamos a continuación el programa en C que realizará estos cálculos.

```
/* raíces reales de una ecuación cuadrática */

#include <stdio.h>
#include <math.h>

main()

{
   float a, b, c, d, x1, x2;

   /* leer datos de entrada */

   printf("a = ");
   scanf("%f", &a);
   printf("b = ");
   scanf("%f", &b);
   printf("c = ");
   scanf("%f", &c);

   /* efectuar los cálculos */

   d = sqrt(b *b - 4 * a *c);
   x1 = (-b + d) / (2 * a);
   x2 = (-b - d) / (2 * a);

   /* escribir salida */

   printf("\nx1 = %e      x2 = %e", x1, x2);
}
```

Este programa no tiene ningún error sintáctico, pero es incapaz de manejar valores negativos para b²-4ac. Es más, se encontrarán dificultades numéricas si la variable a tiene un valor numérico muy pequeño o muy grande, o si a=0. Para cada uno de estos errores se generará un mensaje de error separado.

Supongamos, por ejemplo, que el programa se ejecuta con Turbo C++ con los siguientes valores de entrada:

```
a=1.0    b=2.0    c=3.0
```

El programa se compila sin ninguna dificultad. Sin embargo, cuando se ejecuta el programa objeto se genera el siguiente mensaje de error, después de que se hayan introducido los valores de entrada.

```
sqrt: DOMAIN error
```

Se interrumpe entonces la ejecución del programa, ya que no puede continuar a partir de este punto. La Figura 5.9 ilustra el aspecto de la pantalla en Turbo C++.

Análogamente, supongamos que se ejecuta el programa con los siguientes valores de entrada:

```
a=1E-30        b=1E+10        c=1E+36
```

El sistema ahora generará el siguiente mensaje de error:

```
Floating Point: Overflow
```

cuando se intenta ejecutar el programa. La Figura 5.10 muestra el aspecto de la pantalla en Turbo C++.

5.6. TÉCNICAS DE DEPURACIÓN

Hemos visto que los errores sintácticos y de ejecución suelen conllevar la generación de mensajes de error al compilar o ejecutar el programa. Los errores sintácticos son relativamente fáciles de encontrar y corregir, aun cuando los mensajes de error no sean claros. Los errores de ejecución, por otro lado, son mucho más insidiosos. Cuando ocurre un error de ejecución, debemos en primer lugar determinar su localización (*dónde* ocurre) en el programa. Una vez identificada la localización del error de ejecución, se debe determinar la causa del error (*por qué* ocurre). El conocimiento de dónde ocurre el error ayuda a menudo en el reconocimiento y corrección del error.

Muy relacionados con los errores de ejecución se encuentran los errores *lógicos*. El programa se ejecuta en estos casos de forma correcta, efectuando lo que el programador desea, pero

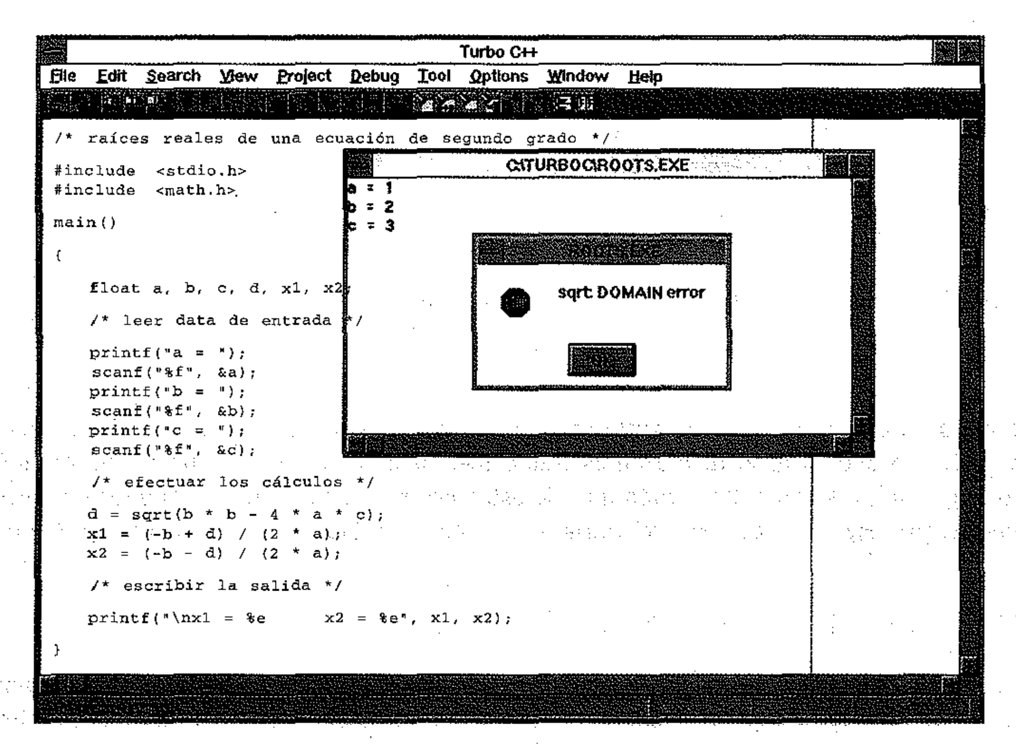


Figura 5.9.

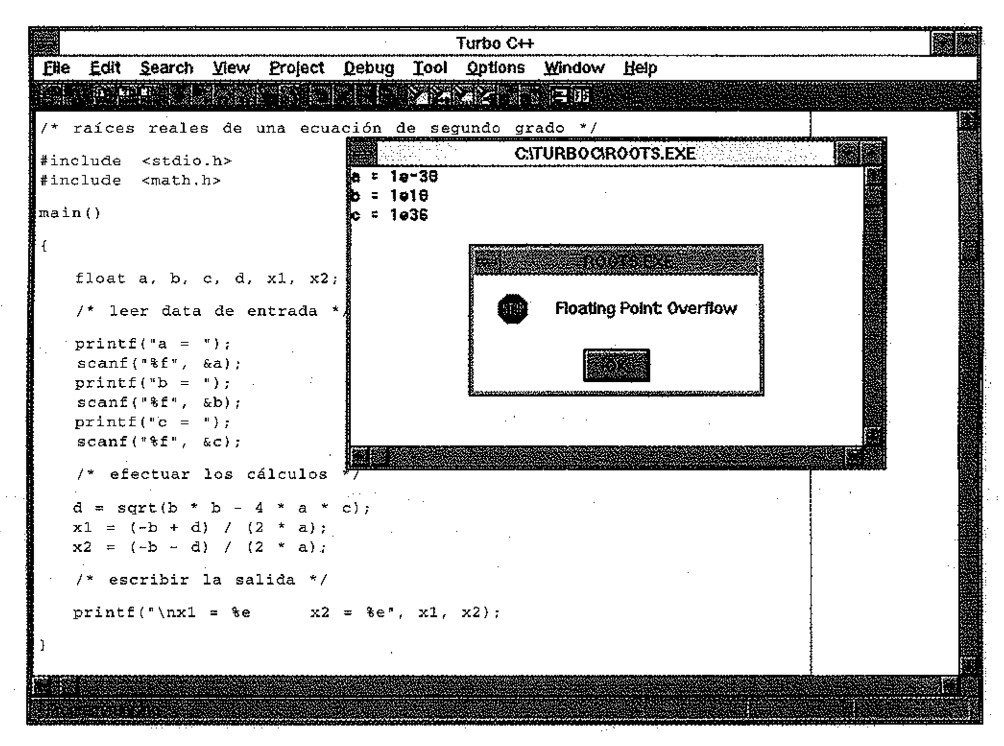


Figura 5.10.

éste ha suministrado a la computadora unas instrucciones que no son lógicamente correctas. Los errores lógicos pueden ser muy difíciles de detectar, puesto que la salida resultante de un programa lógicamente incorrecto puede parecer que está libre de errores. Es más, los errores lógicos son a menudo difíciles de localizar aun cuando se conozca su existencia (por ejemplo, cuando los resultados calculados sean obviamente incorrectos).

Afortunadamente, hay métodos disponibles para encontrar la localización de los errores de ejecución y errores lógicos de un programa. A tales métodos se les denomina *técnicas de depuración*. A continuación se describen algunas de las técnicas de depuración más comúnmente utilizadas.

Aislamiento del error

El aislamiento del error es útil para localizar un error que aparece en un mensaje de error. Si no se conoce la localización general del error, ésta se puede encontrar frecuentemente al borrar una parte del programa con carácter temporal y volver a ejecutar a continuación el programa para ver si el error desaparece. El borrado temporal se realiza rodeando las instrucciones con marcas de comentarios (`/*` y `*/`), convirtiéndolas en comentarios. Si desaparece el mensaje de error, la parte borrada del programa contiene la causa del error.

Una técnica muy relacionada consiste en insertar varias sentencias `printf` tales como

```
printf("Depuración - línea 1\n");

printf("Depuración - línea 2\n");
```

etc.

en diferentes lugares del programa. Cuando se ejecuta el programa, los mensajes de depuración indicarán la localización aproximada del error. Por tanto, la causa del error estará en algún lugar entre la última instrucción `printf` cuyo mensaje *apareció*, y la primera instrucción `printf` cuyo mensaje *no apareció*.

Seguimiento de la traza de ejecución

El seguimiento de la traza de ejecución implica el uso de instrucciones `printf` para visualizar los valores asignados a ciertas variables clave, o para visualizar los valores que se calculan internamente en diferentes lugares del programa. Esta información sirve para diversos propósitos. Por ejemplo, verifica que los valores actualmente asignados a ciertas variables son (o no son) realmente los valores que deberían estar asignados. No es raro el encontrar que los valores asignados actualmente son diferentes de los esperados. Además, esta información permite monitorizar el progreso de los cálculos cuando se ejecuta el programa. En muchas situaciones se podrá localizar el sitio particular donde se producen los fallos debido a que los valores generados serán obviamente incorrectos.

EJEMPLO 5.7. Depuración de un programa. Consideremos de nuevo el programa de cálculo de las raíces reales de una ecuación cuadrática, originalmente mostrado en el Ejemplo 5.6. Vimos que el programa genera el error de ejecución

```
Floating Point: Overflow
```

cuando se ejecuta con los valores de entrada `a = -1E-30, b = 1E10` y `c = 1E36`. Aplicaremos ahora aislamiento del error y técnicas de depuración para determinar la causa del error.

Es razonable suponer que el error se genera en una de las tres instrucciones de asignación siguientes a la última instrucción `scanf`. Por tanto, eliminemos temporalmente estas tres instrucciones colocando delimitadores de comentarios, como se muestra en el listado siguiente.

```
/* raíces reales de una ecuación cuadrática */

#include <stdio.h>
#include <math.h>

main()
{
   float a, b, c, d, x1, x2;

   /* leer datos de entrada */

   printf("a = ");
   scanf("%f", &a);
   printf("b = ");
   scanf("%f", &b);
   printf("c = ");
   scanf("%f", &c);

   /* efectuar los cálculos */
```

```
    /******************** aislamiento del error ***************
    d = sqrt(b *b - 4 * a *c);
    x1 = (-b + d) / (2 * a);
    x2 = (-b - d) / (2 * a);
    ****************** fin aislamiento del error ***************/

    /* escribir salida */

    printf("\nx1 = %e      x2 = %e", x1, x2);
}
```

Cuando se ejecute el programa modificado con los tres valores de entrada el mensaje de error no aparecerá (aunque los valores visualizados para `x1` y `x2` no tendrán sentido). Por tanto, está claro que la causa del error se encuentra en una de estas tres instrucciones.

Eliminaremos a continuación los delimitadores de comentarios, pero precederemos cada instrucción de asignación con una instrucción `printf`, como se muestra a continuación.

```
/* raíces reales de una ecuación cuadrática */

#include <stdio.h>
#include <math.h>

main()

{

   float a, b, c, d, x1, x2;

   /* leer datos de entrada */

   printf("a = ");
   scanf("%f", &a);
   printf("b = ");
   scanf("%f", &b);
   printf("c = ");
   scanf("%f", &c);

   /* efectuar los cálculos */

   printf("Depuración - Línea 1\n"); /* instrucción de depuración temporal */
   d = sqrt(b *b - 4 * a *c);
   printf("Depuración - Línea 2\n"); /* instrucción de depuración temporal */
   x1 = (-b + d) / (2 * a);
   printf("Depuración - Línea 3\n"); /* instrucción de depuración temporal */
   x2 = (-b - d) / (2 * a);

   /* escribir salida */

   printf("\nx1 = %e      x2 = %e", x1, x2);

}
```

Cuando se ejecuta el programa, usando una vez más los mismos valores de entrada, aparecen los tres mensajes de depuración; esto es,

```
Depuración - Línea 1
Depuración - Línea 2
Depuración - Línea 3
```

Concluimos pues que el error de «overflow» ocurrió en la última instrucción de asignación, puesto que dicha instrucción va a continuación de la tercera instrucción `printf`.

Normalmente podríamos terminar aquí nuestros esfuerzos de depuración. Sin embargo, para completar dichas técnicas vamos a eliminar estas tres instrucciones de depuración y las vamos a reemplazar con otras instrucciones `printf` (esto es, tres instrucciones de *seguimiento de la traza de ejecución*). La instrucción sentencia `printf` visualizará los valores de `a`, `b` y `c`, la segunda visualizará el valor de `(-b + d)` y la última el valor de `(-b - d)`, como se muestra a continuación. (Observe la ubicación de las tres instrucciones `printf`, tras el cálculo de `d` pero antes del cálculo de `x1` y `x2`. Observe también los formatos tipo `e` en las instrucciones `printf`.)

```
/* raíces reales de una ecuación cuadrática */

#include <stdio.h>
#include <math.h>

main()

{
   float a, b, c, d, x1, x2;
   /* leer datos de entrada */

   printf("a = ");
   scanf("%f", &a);
   printf("b = ");
   scanf("%f", &b);
   printf("c = ");
   scanf("%f", &c);

   /* efectuar los cálculos */

   d = sqrt(b *b - 4 * a *c);

   /* instrucciones de seguimiento de la traza de ejecución */
   printf("a = %e   b = %e c = %e   d = %e\n", a, b, c, d);
   printf("-b + d = %e\n", (-b + d));
   printf("-b - d = %e\n", (-b - d));

   x1 = (-b + d) / (2 * a);
   x2 = (-b - d) / (2 * a);

   /* escribir salida */

   printf("\nx1 = %e      x2 = %e", x1, x2);
}
```

La ejecución de este programa genera la siguiente salida:

```
a = 1.000000e-30   b = 1.000000e+10   c = 1.000000e+36 d = 1.000000e+10
-b + d = 0.000000e+00
-b - d = -2.000000e+10
```

A partir de estos valores podemos determinar que el valor de `x2` debe ser

```
x2 = (-b - d) / (2 * a) = (-2.000000e+10) / (2 · 1.000000e-30) =
= - 1.000000e+40
```

El valor resultante, `-1.000000e+40`, excede (en tamaño) al número de coma flotante más grande que se puede almacenar en la memoria de la computadora (ver sección 2.4). Por tanto se produce el desbordamiento u «overflow».

La mayoría de los compiladores de C actuales incluyen un *depurador interactivo*, el cual permite establecer *valores de inspección* y *puntos de interrupción*, y posibilita la *ejecución paso a paso*, instrucción a instrucción. Los valores de seguimiento se utilizan habitualmente junto a los puntos de parada o con la ejecución paso a paso, proporcionando una monitorización detallada del programa mientras éste se ejecuta. Estas características son más flexibles y convenientes que las técnicas de aislamiento del error y de seguimiento de la traza de ejecución descritas anteriormente. A continuación describimos con más detalle dichas características.

Valores de inspección

Un *valor de inspección* es el valor de una variable o expresión que se visualiza continuamente durante la ejecución del programa. Por tanto, se pueden ver los cambios producidos en un valor de inspección cuando ocurren como respuesta a la lógica del programa. Al monitorizar unos pocos valores de inspección seleccionados de forma adecuada, podemos determinar a menudo dónde el programa empieza a generar valores incorrectos o no esperados.

En Turbo C++ se pueden definir valores de inspección seleccionando `Add Watch` del menú `Debug` (ver Figura 5.4 más atrás en este capítulo), y especificando a continuación una o más variables o expresiones en el cuadro de diálogo correspondiente. Se visualizarán los valores de inspección en una ventana diferente cuando se ejecute el programa.

Puntos de interrupción

Un *punto de interrupción* es un punto que detiene temporalmente la ejecución de un programa. Al ejecutar el programa, éste se detendrá temporalmente en el punto de interrupción, *antes* de ejecutar la instrucción. Se puede reanudar a continuación la ejecución hasta el siguiente punto de interrupción. Los puntos de interrupción se utilizan junto con los valores de inspección, observando estos últimos en cada punto de parada durante la ejecución del programa.

Para establecer un punto de interrupción en Turbo C++, se selecciona `Add Breakpoint` del menú `Debug` (ver Figura 5.4) y se suministra la información requerida en el cuadro de diálogo resultante. O también, se selecciona una línea particular del programa y se pulsa la tecla de función `F5` para marcarla como un punto de interrupción. Dicho punto de interrupción se puede

desactivar más adelante volviendo a pulsar `F5`. (La tecla de función `F5` actúa como un «conmutador» en este contexto, puesto que al pulsarla se activa o desactiva el punto de interrupción).

Ejecución paso a paso

La *ejecución paso a paso* significa ejecutar una instrucción en cada momento, normalmente pulsando una tecla de función para ejecutar dicha instrucción. En Turbo C++, por ejemplo, la ejecución paso a paso se puede efectuar pulsando o la tecla de función `F7` o bien `F8`. (`F8` ejecuta las funciones subordinadas de una vez, mientras que `F7` las ejecuta paso a paso.) Al ejecutar paso a paso un programa completo, se puede determinar qué instrucciones producen resultados erróneos o generan mensajes de error.

La ejecución paso a paso se utiliza frecuentemente con los valores de inspección, permitiendo rastrear la historia completa de un programa cuando éste se ejecuta. De esta forma, se pueden observar cambios en los valores de inspección cuando éstos ocurren. Esto nos permite determinar las instrucciones que generan resultados erróneos.

EJEMPLO 5.8. Depuración con un depurador interactivo. Consideremos de nuevo el programa presentado en los Ejemplos 5.6 y 5.7, de cálculo de las raíces de una ecuación cuadrática. Utilizaremos ahora el depurador interactivo de Turbo C++ para determinar la causa del error cuando se ejecuta el programa con los valores de entrada `a = 1E-30`, `b = 1E10` y `c = 1E36`.

La Figura 5.11 muestra el programa dentro de la ventana de edición de Turbo C++. Se han seleccionado tres valores de inspección para las cantidades `-b+d`, `-b-d` y `2*a`. Cada valor de inspección fue seleccionado eligiendo `Add Watch` del menú `Debug`. En la ventana `Watch`, sobre el listado del programa, se encuentran los valores de inspección.

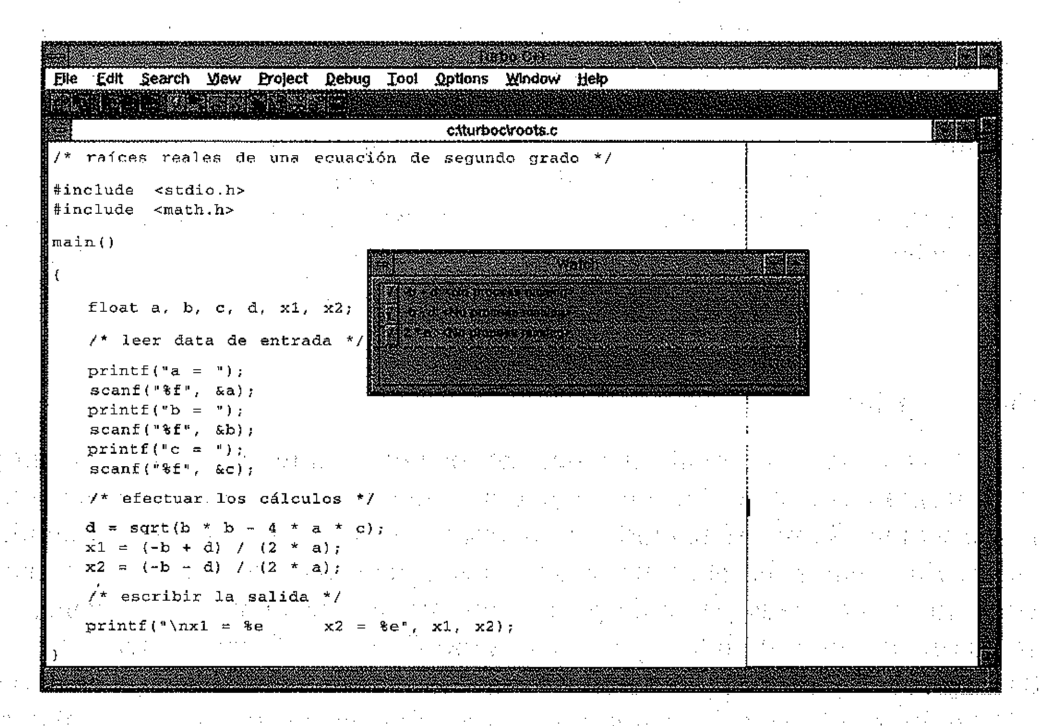


Figura 5.11.

Se ha definido además un punto de interrupción en la primera instrucción de asignación, es decir, `d = sqrt(b*b - 4*a*c)`. El punto de interrupción se ha definido situando el cursor sobre la instrucción deseada y pulsando a continuación la tecla `F5`. En la Figura 5.11 se muestra destacado el punto de interrupción.

Observe que la Figura 5.11 muestra el estado del programa *antes* de comenzar su ejecución. Ésta es la razón por la que aparece el mensaje `<No process running>` al lado de cada valor de inspección.

Una vez que comienza la ejecución del programa (al seleccionar `Run` del menú `Debug`), se introducen los valores de `a`, `b` y `c` por teclado y continúa la ejecución hasta el siguiente punto de interrupción. El programa se detiene temporalmente, como se muestra en la Figura 5.12. Observe que todavía no se ha ejecutado la primera instrucción de asignación, de manera que `d` no tiene aún un valor asignado. Por tanto, los dos primeros valores de inspección están aún sin definir. Sin embargo, el último valor de inspección se obtiene directamente de los datos de entrada. Su valor, `2e-30`, se muestra en la ventana de inspección de la Figura 5.12.

Podemos reanudar la ejecución, continuando hasta el final del programa, seleccionando de nuevo `Run` del menú `Debug`. Sin embargo, en su lugar vamos a ejecutar paso a paso el programa pulsando la tecla de función `F8` dos veces. La Figura 5.13 muestra el estado del programa en este punto. Observe que el punto de interrupción permanece destacado. Además, también está destacada la tercera instrucción de asignación (es decir, `x2 = (-b -d) / (2 * a))`. Esta última es la siguiente instrucción a ejecutar.

En la ventana de inspección podemos ver ahora los valores actuales de todas las variables de inspección. Fácilmente se observa que el valor asignado a `x2`, el cual es el cociente entre el segundo y tercer valor de inspección, producirá un desbordamiento. De este modo, si reanudamos la ejecución del programa, bien seleccionando `Run` del menú `Debug` o bien mediante la ejecución paso a paso, aparecerá el mensaje de «overflow» mostrado en la Figura 5.10.

Algunas veces no se puede localizar un error, aun utilizando las técnicas de depuración más sofisticadas. En tales casos, los programadores principiantes se inclinan a menudo a sospechar

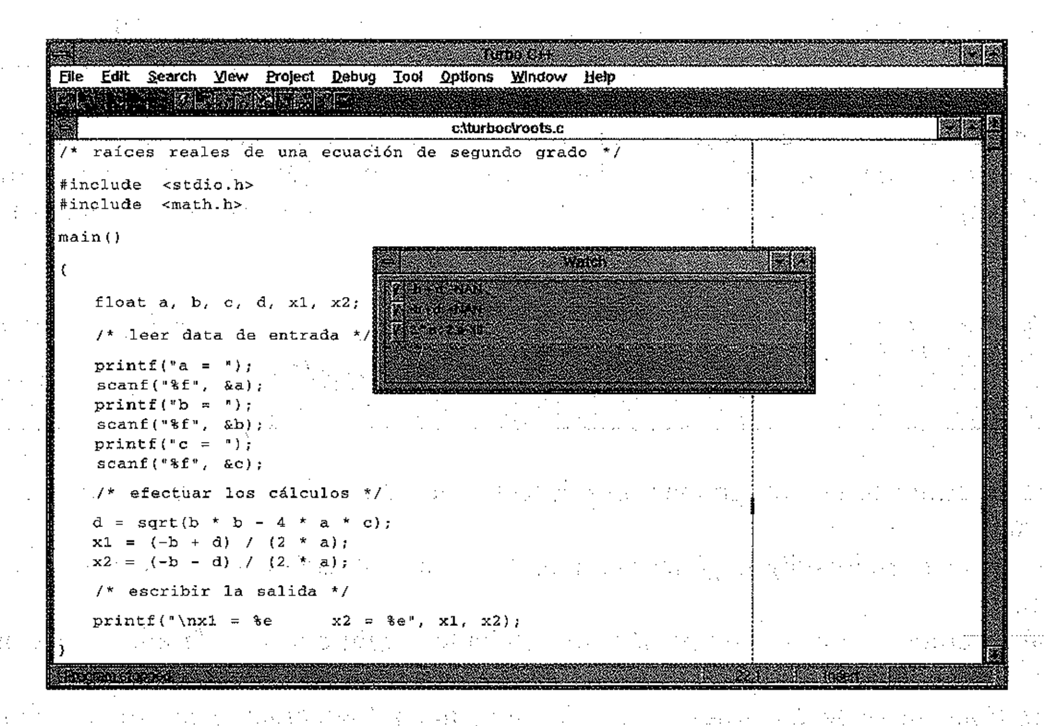


Figura 5.12.

Turbo C++
File Edit Search View Project Debug Tool Options Window Help
c:\turboc\roots.c

```
/* raíces reales de una ecuación de segundo grado */

#include  <stdio.h>
#include  <math.h>

main()
{
   float a, b, c, d, x1, x2;

   /* leer data de entrada */

   printf("a = ");
   scanf("%f", &a);
   printf("b = ");
   scanf("%f", &b);
   printf("c = ");
   scanf("%f", &c);

   /* efectuar los cálculos */

   d = sqrt(b * b - 4 * a * c);
   x1 = (-b + d) / (2 * a);
   x2 = (-b - d) / (2 * a);

   /* escribir la salida */

   printf("\nx1 = %e     x2 = %e", x1, x2);
}
```

Figura 5.13.

que el problema está más allá de su control, como un error del hardware o un error del compilador. Sin embargo, el problema casi siempre resulta ser un sutil error de la lógica del programa. En tales situaciones, debemos resistir la tentación de echarle las culpas a la computadora y no buscar el escurridizo error de programación. Aunque los errores de las computadoras ocurren *raras veces*, éstos normalmente producen resultados muy peculiares, tales como el bloqueo de la computadora o visualización de caracteres aleatorios e ininteligibles.

Finalmente, hemos de reconocer que al programar no nos escaparemos casi nunca de cometer algunos errores lógicos, por muy cuidadosos que seamos. Debemos por tanto anticipar la necesidad de alguna depuración lógica cuando escribamos programas reales y significativos.

CUESTIONES DE REPASO

5.1. ¿Qué se entiende por programación «descendente»? ¿Cuáles son sus ventajas? ¿Cómo se efectúa?

5.2. ¿Qué es el pseudocódigo? ¿Qué ventajas tiene su utilización al planificar un nuevo programa?

5.3. ¿Qué se entiende por programación «ascendente»? ¿En qué se diferencia de la programación descendente?

5.4. ¿Qué libertad tiene un programador al secuenciar lógicamente las instrucciones en un programa en C? Explicarlo.

5.5. ¿Por qué se sangran algunas instrucciones en un programa en C? ¿Es absolutamente necesario el sangrado de estas líneas?
5.6. ¿Qué razones existen para incluir comentarios en un programa en C? ¿Qué extensión deben tener estos comentarios?
5.7. Citar dos factores que contribuyan a la generación de datos de salida de forma clara y legible.
5.8. ¿Qué información de utilidad proporcionan los rótulos?
5.9. ¿Cómo se introduce un programa en la computadora en la mayoría de los entornos de programación en C actuales?
5.10. ¿Qué es la extensión del nombre de un programa?
5.11. ¿Qué es un error sintáctico? Citar algunos errores sintácticos comunes.
5.12. ¿Qué es un error de ejecución? Citar algunos errores de ejecución comunes.
5.13. ¿En qué se diferencian los errores sintácticos de los de ejecución?
5.14. ¿Qué es un error lógico? ¿En qué se diferencian los errores lógicos de los sintácticos y los de ejecución?
5.15. ¿Qué son los mensajes de error?
5.16. ¿Qué diferencia hay entre los mensajes de error de compilación y de ejecución? Citar algunas situaciones en las que se genere cada tipo de mensaje de error.
5.17. ¿Qué se entiende por aislamiento del error? ¿Para qué sirve? ¿Cómo se efectúa?
5.18. ¿Qué se entiende por inspección de la traza de ejecución? ¿Para qué sirve? ¿Cómo se realiza?
5.19. ¿Qué es un depurador interactivo? ¿De qué características especiales dispone un depurador?
5.20. ¿Qué son los valores de inspección? ¿Para qué se utilizan? ¿Cómo se definen en términos generales?
5.21. ¿Qué son los puntos de interrupción? ¿Para qué se utilizan? ¿Cómo se definen en términos generales?
5.22. ¿En qué consiste la ejecución paso a paso? ¿Para qué se utiliza? ¿Cómo se efectúa en términos generales?
5.23. Describir cómo se pueden utilizar los valores de inspección con los puntos de interrupción y la ejecución paso a paso para monitorizar el avance en la ejecución de un programa.

PROBLEMAS

Las siguientes cuestiones están más relacionadas con la comprensión del capítulo realmente que con la resolución de problemas.

5.24. Obtener respuestas a las siguientes preguntas relativas a las computadoras personales de la escuela u oficina.

a) ¿De qué equipamiento se dispone exactamente (impresoras, dispositivos de memoria auxiliar, etc.)?
b) ¿De qué sistemas operativos se dispone?

c) ¿Cómo se pueden grabar, visualizar y transferir archivos (programas) de un dispositivo de memoria a otro?
d) ¿Cuál es el coste aproximado de una computadora personal?

5.25. Obtener respuestas a las siguientes preguntas relativas al compilador de C de la escuela u oficina.

a) ¿De qué versión de C se dispone? ¿Qué sistema operativo requiere?
b) ¿Cómo se accede al compilador de C? ¿Cómo se accede a un programa en C una vez activado el compilador? ¿Cómo se visualiza un programa? ¿Cómo se graba?
c) ¿Cómo se efectúan las funciones de edición usuales (insertar, borrar, etc.)?
d) ¿Cómo se compila y ejecuta un programa en C?
e) ¿Incluye el compilador un depurador interactivo? En caso afirmativo, ¿qué características soporta el depurador? ¿Cuáles son las características más comúnmente utilizadas?
f) ¿Cuál es el coste del compilador de C?

PROBLEMAS DE PROGRAMACIÓN

5.26. En el Ejemplo 1.6 se presenta un programa en C para calcular el área de un círculo, dado su radio. Introducir este programa en la computadora y hacer las modificaciones necesarias, tales como `#include <stdio.h>`. Asegurarse de corregir cualquier error tipográfico. Listar el programa una vez que se haya almacenado en la computadora. Cuando se esté seguro de su corrección, compilar el programa y ejecutar a continuación el programa objeto introduciendo diferentes valores del radio. Verificar que las respuestas calculadas son correctas comparándolas con las calculadas a mano.

5.27. Introducir, compilar y ejecutar los programas en C dados en los Ejemplos 1.7 a 1.13. Verificar que se ejecutan correctamente con la versión particular de C. (Si cualquiera de los programas no funciona, intentar determinar por qué.)

5.28. Repetir el Problema 5.27 para algunos de los programas dados en el Problema 1.31.

5.30. Escribir un programa completo en C para cada uno de los puntos siguientes. Introducir cada programa en la computadora, asegurándose de haber corregido cualquier error tipográfico. Guardar el programa cuando estemos seguros de haberlo introducido correctamente, y compilarlo y ejecutarlo a continuación. Asegurarse de incluir rótulos para todas las entradas, y etiquetar toda la salida.

a) Escribir `¡HOLA!` al comienzo de una línea.
b) Hacer que la computadora escriba

```
HOLA, ¿COMO TE LLAMAS?
```

en una línea. El usuario introducirá a continuación su nombre. La computadora escribe dos líneas en blanco y a continuación

```
BIENVENIDO (nombre)
¡SEAMOS AMIGOS!
```

en dos líneas consecutivas. Utilice un vector de caracteres para representar el nombre del usuario. Suponga que el nombre contiene memos de 20 caracteres.

c) Convertir una temperatura leída en grados Fahrenheit a grados Celsius utilizando la fórmula

$C = (5/9) \times (F - 32)$

Comprobar el programa con los siguientes valores: 68, 150, 212, 0, –22, –200 (grados Fahrenheit).

d) Determinar cuánto dinero (en dólares) hay en una hucha que contiene varios medios dólares, cuartos, dimes, níqueles y peniques. Utilizar los siguientes valores para comprobar el programa: 11 medios dólares, 7 cuartos, 3 dimes, 12 níqueles y 17 peniques. (*Respuesta*: 8,32 dólares).

e) Calcular el volumen y el área de una esfera utilizando las fórmulas

$V = 4\pi r^3/3$
$A = 4\pi r^2$

Comprobar el programa utilizando los siguientes valores del radio: 1, 6, 12,2, 0,2.

f) Calcular la masa de aire de un neumático de automóvil utilizando la fórmula

$PV = 0.37m(T + 460)$

en donde:

P = presión, libras por pulgada cuadrada (psi)
V = volumen, pies cúbicos
m = masa de aire, libras
T = temperatura, grados Fahrenheit

El neumático contiene dos pies cúbicos de aire. Supóngase que la presión es 32 psi a la temperatura ambiente.

g) Introducir una palabra de cinco letras en la computadora, codificar a continuación la palabra letra a letra restando 30 del valor numérico que se utiliza para representar cada letra. Por tanto, si se está utilizando el juego de caracteres ASCII, la letra `a` (que se representa con el valor 97) se transformará en `c` (representada por el valor 67), etc.

Escribir la versión codificada de la palabra. Comprobar el programa con las siguientes palabras: negro, rosas, Japón, Cebra.

h) Leer una palabra de cinco letras que se haya codificado de la forma indicada en el apartado anterior. Decodificar la palabra y escribir la palabra decodificada.

i) Leer una línea de texto completa y codificarla utilizando el método descrito anteriormente. Visualizar la línea completa de texto en la forma codificada. Decodificar a continuación el texto y escribirlo (visualizar el texto como apareció originalmente) utilizando el método descrito en el apartado *h*).

j) Leer una línea de texto que contenga letras mayúsculas y minúsculas. Escribir el texto con letras mayúsculas y minúsculas intercambiadas y el resto de los caracteres intactos. (*Sugerencia:* Utilizar el operador condicional `?:` y las funciones de biblioteca `islower`, `tolower` y `toupper`.)

CAPÍTULO 6

Instrucciones de control

En la mayoría de los programas en C que hemos encontrado hasta el momento, las instrucciones se ejecutaban en el mismo orden en que aparecían en el programa. Cada instrucción se ejecutaba una única vez. Los programas que se suelen escribir en la práctica no son tan sencillos como éstos, ya que los que hemos visto no incluyen ningún tipo de elementos de control lógico. En particular, en estos programas no aparecen comprobaciones de condiciones que sean verdaderas o falsas ni aparece la ejecución repetida de grupos individuales de instrucciones de forma selectiva. La mayoría de los programas de interés práctico hacen uso intenso de estos elementos.

Por ejemplo, muchos programas requieren que se efectúe una comprobación lógica en algún punto concreto de los mismos. A continuación se realizará alguna acción que dependerá del resultado del test lógico. Esto se conoce como *ejecución condicional*. Existe también una clase especial de ejecución condicional, llamada *selección*, en la que se selecciona un grupo de instrucciones entre varios grupos disponibles. Además, los programas pueden requerir que un grupo de instrucciones se ejecute repetidamente, hasta que se satisfaga alguna condición lógica. Esto se conoce por el nombre de *bucle*. A veces se conoce por adelantado el número de repeticiones requeridas; y otras veces el cálculo continúa indefinidamente hasta que la condición lógica se hace verdadera.

Se pueden realizar todas estas operaciones utilizando diversas instrucciones de control incluidas en C. Veremos cómo hacer esto en este capítulo. El uso de estas instrucciones nos abrirá las puertas a problemas de programación mucho más amplios e interesantes que los vistos hasta ahora.

6.1. INTRODUCCIÓN

Antes de considerar con detalle las instrucciones de control de las que se dispone en C, revisemos algunos conceptos presentados en los Capítulos 2 y 3 que se deben utilizar en conjunción con estas instrucciones. Comprender estos conceptos es fundamental para lo que sigue.

Primero necesitaremos formar expresiones lógicas que serán verdaderas o falsas. Para hacer esto podemos utilizar los cuatro operadores relacionales, <, <=, >, >=, y los dos operadores de igualdad, == y != (ver sección 3.3).

EJEMPLO 6.1. A continuación se muestran varias expresiones lógicas.

```
cont <= 100

sqrt(a+b+c) > 0.005
```

```
respuesta == 0

balance >= maximo

car1 < 'T'

letra != 'x'
```

Las cuatro primeras expresiones involucran operandos numéricos. Su significado debe resultar fácilmente comprensible.

En la quinta expresión, `car1` se supone que es una variable de tipo carácter. Esta expresión será verdadera si el carácter representado por `car1` se encuentra antes de la `T` en el juego de caracteres, esto es, si el valor numérico que se utiliza para codificar el carácter es menor que el valor numérico utilizado para codificar la letra `T`.

La última expresión hace uso de la variable de tipo carácter `letra`. Esta expresión será verdadera si el carácter representado por `letra` es algún otro que no sea `x`.

Además de los operadores relacionales y de igualdad, C posee dos *conectivas lógicas* (también llamadas *operadores lógicos*), `&&` (Y) y `||` (O), y el operador unario de negación `!` (ver sección 3.3). Las conectivas lógicas se utilizan para combinar expresiones lógicas, formándose así expresiones más complejas. El operador de negación se utiliza para invertir el valor de una expresión lógica (por ejemplo, de verdadero a falso).

EJEMPLO 6.2. A continuación se muestran algunas expresiones lógicas que ilustran la utilización de conectivas lógicas y del operador de negación.

```
(cont <= 100) && (car1 != '*')

(balance < 1000.0) || (estado == 'R')

(respuesta < 0) || ((respuesta > 5.0) && (respuesta <= 10.0))

!((pagos >= 1000.0) && (estado == 's'))
```

Observe que en estos ejemplos `car1` y `estado` se suponen variables de tipo carácter. Las otras variables se suponen numéricas (enteras o en coma flotante).

Ya que los operadores relacionales y de igualdad tienen mayor precedencia que los operadores lógicos, algunos de los paréntesis no son necesarios en las expresiones anteriores (ver Tabla 3.1 en la sección 3.5). Por tanto, podríamos haber escrito estas expresiones como sigue:

```
cont <= 100 && car1 != '*'

balance < 1000.0 || estado == 'R'

respuesta < 0 || respuesta > 5.0 && respuesta <= 10.0

!(pagos >= 1000.0 && estado == 's')
```

Sin embargo, suele ser una buena idea incluir paréntesis si puede haber duda en la precedencia de los operadores. Esto es particularmente cierto en expresiones relativamente complicadas, tal como la expresión tercera de las anteriores.

El *operador condicional* `? :` también hace uso de una expresión que es verdadera o falsa (ver sección 3.5). Se selecciona un valor en función del resultado de esta expresión lógica. Este operador es equivalente a la estructura *if-then-else* sencilla (ver sección 6.6).

EJEMPLO 6.3. Supongamos que `estado` es una variable de tipo carácter y `balance` una variable en coma flotante. Deseamos asignar el carácter `C` a `estado` si `balance` tiene el valor cero, y `0` si `balance` tiene un valor mayor que cero. Esto se puede hacer escribiendo

```
estado = (balance == 0) ? 'C' : '0'
```

Finalmente, recordemos que existen tres clases diferentes de instrucciones en C: *instrucciones de expresión*, *instrucciones compuestas* e *instrucciones de control* (ver sección 2.8). Una instrucción de expresión consiste en una expresión seguida de un punto y coma (ver sección 2.7). Una instrucción compuesta consiste en una secuencia de dos o más instrucciones consecutivas encerradas entre llaves ({ y }). Las instrucciones incluidas en las llaves pueden ser instrucciones de expresión, otras instrucciones compuestas o instrucciones de control. La mayoría de las instrucciones de control contienen instrucciones de expresión o instrucciones compuestas, incluyendo las instrucciones compuestas anidadas.

EJEMPLO 6.4. A continuación se muestra una instrucción compuesta que se utilizó en el Ejemplo 3.31.

```
{
   int minusc, mayusc;

   minusc = getchar();
   mayusc = toupper(minusc);
   putchar(mayusc);
}
```

He aquí una instrucción compuesta más complicada:

```
{

   float sum = 0, sumsq = 0, sumsqrt = 0, x;

   scanf("%f", &x);
   while (x != 0)  {
         sum += x;
         sumsq += x*x;
         sumsqrt += sqrt(x);
         scanf("%f", &x);

   }

}
```

Este último ejemplo contiene una instrucción compuesta anidada en otra.

Las instrucciones de control presentadas en este capítulo hacen uso frecuente de expresiones lógicas e instrucciones compuestas. También se utilizan *operadores de asignación*, tales como el utilizado en el ejemplo anterior (+=).

6.2. EJECUCIÓN CONDICIONAL: LA INSTRUCCIÓN `if-else`

La instrucción `if-else` se utiliza para realizar una comprobación lógica y a continuación llevar a cabo una de dos posibles acciones, dependiendo del resultado de la comprobación (de que sea verdadera o falsa).

La parte `else` de la instrucción `if-else` es opcional. Por tanto, las instrucciones se pueden escribir, en su forma general más simple,

```
if (expresión) instrucción
```

La *`expresión`* se debe encontrar entre paréntesis, como se ha indicado. De esta forma, la *`instrucción`* se ejecutará sólo si la *`expresión`* tiene un valor no nulo (si *`expresión`* es verdadera). Si la *`expresión`* tiene el valor cero (si *`expresión`* es falsa), entonces se ignorará la *`instrucción`*.

La *`instrucción`* puede ser simple o compuesta. En la práctica suele ser una instrucción compuesta que puede incluir otras instrucciones de control.

EJEMPLO 6.5. A continuación se muestran varias instrucciones `if` representativas.

```
if (x < 0) printf("%f", x);

if (debito > 0)
   credito = 0;

if (x <= 3.0) {
   y = 3 * pow(x, 2);
   printf("%f\n", y);
}

if ((balance < 1000.) || (estado == 'R'))
   printf("%f", balance);

if ((a >= 0) && (b <= 5)) {
   xmid = (a + b) / 2;
   ymid = sqrt(xmid);
}
```

La primera instrucción hace que se imprima (visualice) el valor de la variable en coma flotante `x` si su valor es negativo. En la segunda instrucción, se asigna el valor cero a `credito` si el valor de `debito` es mayor que cero. La tercera instrucción involucra una instrucción compuesta en la que se evalúa `y` y a continuación se visualiza, si el valor de `x` no excede de `3`. En la cuarta instrucción tenemos una expresión lógica compleja, que hace que el valor de `balance` se visualice si su valor es menor de `1000` o si `estado` tiene asignado el valor `'R'`.

La última instrucción involucra una expresión lógica compleja y una instrucción compuesta. A las variables `xmid` e `ymid` se les asignarán, pues, valores apropiados sólo si el valor actual de `a` es no negativo *y* el valor actual de `b` no excede de `5`.

La forma general de una instrucción `if` que incluye la cláusula `else` es

`if (expresión) instrucción 1 else instrucción 2`

Si la *expresión* tiene un valor no nulo (si *expresión* es verdadera), entonces se ejecutará *instrucción 1*. En otro caso (*expresión* es falsa), se ejecutará *instrucción 2*.

EJEMPLO 6.6. A continuación presentamos varios ejemplos como muestra del uso de la instrucción `if - else` completa.

```
if (estado == 'S')
   tasa = 0.20 * pago;
else
   tasa = 0.14 * pago;

if (debito > 0) {
   printf("cuenta nº %d está en números rojos", no_cuenta);
   credito = 0;
}
else
   credito = 1000.0;

if (x <= 3)
   y = 3 * pow(x, 2);
else
   y = 2 * pow((x - 3), 2);
printf("%f\n", balance);

if (circulo) {
   scanf("%f", &radio);
   area = 3.14159 * radio * radio;
   printf("Area del círculo = %f", area);
}
else {
   scanf("%f %f", &longitud, &anchura);
   area = longitud * anchura;
   printf("Area del rectángulo = %f", area);
}
```

En el primer ejemplo el valor de `tasa` se determina de dos formas posibles, dependiendo del carácter que se le haya asignado a la variable `estado`. Observe el punto y coma al final de cada instrucción, especialmente en la primera (`tasa = 0.20 * pago;`). Una forma más concisa de hacer lo mismo es la siguiente:

```
tasa = (estado == 'S') ? (0.20 * pago) : (0.14 * pago);
```

si bien este estilo no queda tan claro.

El segundo ejemplo examina el estado de débito de una cuenta. Si el valor de `debito` es mayor que cero, se visualiza un mensaje y el límite de crédito se establece en cero; en cualquier otro caso, el límite de crédito se establece en `1000.0`. En el tercer ejemplo se calcula el valor de `y` de dos formas distintas, dependiendo de si el valor de `x` es mayor o no que `3`.

El cuarto ejemplo muestra cómo se puede calcular un área de dos figuras geométricas. Si a `circulo` se le asigna un valor no nulo, se lee el radio del círculo, se calcula el área y luego se visualiza. Pero si el valor de `circulo` es cero, entonces se leen la longitud y la anchura de un rectángulo, se calcula el área y luego se visualiza. En cada caso, se rotula el tipo de figura geométrica junto al valor del área correspondiente.

Es posible *anidar* (incluir) instrucciones `if - else` una dentro de otra. Estas anidaciones pueden tomar formas diferentes. La forma más general de anidamiento doble es

```
if e1 if e2 s1
      else s2
else  if e3 s3
      else s4
```

en donde *e1*, *e2* y *e3* representan expresiones lógicas y *s1*, *s2*, *s3* y *s4* instrucciones. En esta situación se ejecutará una instrucción `if - else` completa si *e1* es no nula (verdadera), y s ejecutará otra instrucción `if - else` si *e1* es cero (falsa). Es posible, por supuesto, que *s1*, *s2*, *s3* y *s4* contengan otras instrucciones `if - else`. Tendríamos en este caso un anidamiento múltiple.

Algunas otras formas de anidamiento doble son

```
if e1 s1
else if e2 s2

if e1 s1
else if e2 s2
     else s3

if e1 if e2 s1
      else s2
else s3

if e1 if e2  s1
      else s2
```

En los tres primeros casos la asociación entre las cláusulas `else` y sus expresiones correspondientes es clara. Sin embargo, en el último caso no queda claro qué expresión (*e1* o *e2*) tiene asociada la cláusula `else`. La respuesta es *e2*. La regla a aplicar es que la cláusula `else` siempre se encuentra asociada al `if` no aparejado (sin `else`) precedente más cercano. El sangrado de las líneas debe sugerir esto, aunque no es éste el factor decisivo. Por tanto, el último ejemplo es equivalente a

```
if e1  {
   if e2 s1 else s2
}
```

Si quisiéramos asociar la cláusula `else` a *e1* en lugar de a *e2*, podríamos hacer esto escribiendo

```
if e1  {
   if e2 s1
}
else s2
```

Este tipo de anidamiento se debe utilizar con cuidado, con vistas a evitar posibles ambigüedades.

En algunas situaciones puede resultar interesante anidar múltiples instrucciones `if - else`, para crear una situación en la que se seleccionará una entre varias acciones. Por ejemplo, la forma general de cuatro instrucciones `if - else` anidadas se puede escribir así:

```
if e1 s1
else if e2 s2
     else if e3 s3
          else if e4 s4
               else s5
```

Cuando se encuentre una expresión lógica con valor no cero (verdadera), se ejecutará la correspondiente instrucción y no se evaluarán el resto de las instrucciones `if - else`. De esta forma, el control se transfiere fuera de toda la anidación una vez que se encuentre una condición verdadera.

La cláusula `else` final se aplicará sólo si ninguna de las expresiones es verdadera. Se puede utilizar la misma para proporcionar una condición por omisión o un mensaje de error.

EJEMPLO 6.7. A continuación se muestra un ejemplo de tres instrucciones `if - else` anidadas.

```
((hora >= 0.) && (hora < 12.))
if printf("Buenos Días");
((hora >= 12.) && (hora < 18.))
else if printf("Buenas Tardes");
((hora >= 18.) && (hora < 24.))
else if printf("Buenas Noches");
else printf("Hora no válida");
```

Este ejemplo hace que se visualice un mensaje diferente a distintas horas del día. Concretamente, se visualizará `Buenos Días` si `hora` tiene un valor entre `0` y `12`; se visualizará `Buenas Tardes` si `hora` tiene un valor entre `12` y `18`; y se visualizará `Buenas Noches` si `hora` tiene un valor entre `18` y `24`. Si *hora* tiene un valor menor que cero, o mayor o igual a 24, se visualizará un mensaje de error (`"Hora no válida"`).

6.3. BUCLES: LA INSTRUCCIÓN `while`

La instrucción `while` se utiliza para generar bucles, en los cuales un grupo de instrucciones se ejecuta de forma repetida, hasta que se satisface alguna condición.

La forma general de la instrucción `while` es

```
while (expresión) instrucción
```

La *instrucción* se ejecutará repetidamente, mientras el valor de *expresión* sea verdadero (mientras *expresión* tenga un valor no nulo). Esta *instrucción* puede ser simple o compuesta, aunque suele ser compuesta. Debe incluir algún elemento que altere el valor de *expresión*, proporcionando así la condición de salida del bucle.

EJEMPLO 6.8. Cantidades enteras consecutivas. Supongamos que queremos visualizar los dígitos 0, 1, 2, ..., 9, apareciendo cada uno en cada línea. Esto se puede hacer con el siguiente programa.

```
#include <stdio.h>

main()     /* visualizar los números del 0 al 9 */

{
   int digito = 0;

   while (digito <= 9)  {
          printf("%d\n", digito);
          ++digito;
   }
}
```

Inicialmente se le asigna a `digito` el valor `0`. El bucle `while` visualiza a continuación el valor actual de `digito`, incrementa su valor en `1` y repite el ciclo hasta que el valor de `digito` sea mayor que `9`. El efecto total es que el cuerpo del bucle se repetirá `10` veces, generándose `10` líneas consecutivas de salida. Cada línea contendrá un número, comenzando por `0` y acabando en `9`. Por tanto, cuando se ejecute el programa, se genera la siguiente salida.

```
0
1
2
3
4
5
6
7
8
9
```

Este programa se puede escribir de forma más concisa como se presenta a continuación.

```
#include <stdio.h>

main()     /* visualizar los números del 0 al 9 */
{
   int digito = 0;

   while (digito <= 9)
          printf("%d\n", digito++);
}
```

Cuando se ejecute, este programa generará la misma salida que el primero.

En algunos bucles en particular, se conoce *a priori* el número de pasadas a través del bucle. El ejemplo anterior muestra un bucle de este tipo. Sin embargo, en otras ocasiones no se conoce por adelantado el número de pasadas a través del bucle. En su lugar, la repetición continúa de forma indefinida, hasta que se satisfaga una condición lógica especificada. Para este segundo tipo de bucles es bastante adecuada la instrucción `while`.

EJEMPLO 6.9. Conversión de minúsculas a mayúsculas. En este ejemplo se lee una línea de texto en minúsculas carácter a carácter y se almacenan los caracteres en un array llamado `letras`. El programa continúa leyendo caracteres hasta que se lea un carácter de fin de línea (`EOL`). A continuación se transforman los caracteres a mayúsculas, utilizando la función de biblioteca `toupper`, y se escriben.

Se utilizan dos instrucciones `while` por separado. La primera lee el texto introducido por teclado. Obsesrve que no se conoce *a priori* el número de pasadas a través de este bucle. El segundo bucle `while` realiza la conversión y escribe el texto transformado. Efectuará un número conocido de pasadas, dado que el número de caracteres a escribir viene determinado al contar el número de pasadas a través del primer bucle.

A continuación se muestra el programa completo.

```
/* convertir una línea de texto de minúsculas a mayúsculas */

#include <stdio.h>
#include <ctype.h>

#define EOL '\n'

main()
{
   char letras[80];
   int aux, cont = 0;

   /* leer el texto en minúsculas */
   while ((letras[cont] = getchar()) != EOL) ++cont;
   aux = cont;

   /* escribir el texto en mayúsculas */
   cont = 0;
   while (cont < aux)  {
          putchar(toupper(letras[cont]));
          ++cont;
   }
}
```

Observe que `cont` tiene asignado inicialmente el valor cero. Su valor se incrementa en `1` en cada pasada por el primer bucle. El valor final de `cont`, al concluir el primer bucle, se asigna entonces a `aux`. El valor de `aux` determina el número de pasadas por el segundo bucle.

El primer bucle `while`, esto es,

```
while ((letras[cont] = getchar()) != EOL) ++cont;
```

está escrito de forma muy concisa. Este bucle de una sola instrucción es equivalente al siguiente:

```
letras[cont] = getchar();
while (letras[cont] != EOL)  {
      cont = cont + 1;
      letras[cont] = getchar();
}
```

Esta última forma resultará más familiar a aquellos lectores que hayan utilizado otros lenguajes de programación de alto nivel, como Pascal o BASIC. Las dos formas son correctas, si bien la primera es más representativa del estilo de programación típico en C.

Cuando se ejecuta el programa, cualquier línea de texto que se introduzca se visualizará a continuación en mayúsculas. Supongamos, por ejemplo, que se introduce la siguiente línea de texto:

```
tu mirada me recuerda el fulgor de la aurora...
```

La computadora responderá escribiendo

```
TU MIRADA ME RECUERDA EL FULGOR DE LA AURORA...
```

EJEMPLO 6.10. Media de una lista de números. Utilicemos ahora una instrucción `while` para calcular la media de una lista de `n` números. Basaremos nuestra estrategia en el uso de una suma parcial que inicialmente sea igual a cero, actualizando este valor cada vez que se introduzca un número en la computadora. Por tanto, el problema sugiere de forma muy natural el uso de un bucle.

Los cálculos se efectuarán de la siguiente manera.

1. Asignar el valor de `1` a la variable entera `cont`. Esta variable se utilizará como contador en el bucle.
2. Asignar el valor de `0` a la variable en coma flotante `suma`.
3. Leer el valor de la variable entera `n`.
4. Realizar los siguientes pasos de forma repetida mientras `cont` no sea mayor que `n`.
   a) Leer un número de la lista. Cada número se almacenará en la variable en coma flotante `x`.
   b) Añadir el valor de `x` al actual de `suma`.
   c) Incrementar en `1` el valor de `cont`.
5. Dividir por `n` el valor de `suma` para obtener el deseado de la media.
6. Escribir el valor calculado de la media.

Presentamos a continuación el programa en C. Nótese que las operaciones de entrada van acompañadas de rótulos que piden al usuario la información requerida.

```
/* calcular la media de n números */

#include <stdio.h>

main()

{
   int n, cont = 1;
   float x, media, suma = 0;
```

```
    /* inicializar y leer el valor de n */
    printf("¿Cuantos números? ");
    scanf("%d", &n);

    /* leer los números */
    while (cont <= n)  {
          printf("x = ");
          scanf("%f", &x);
          suma += x;
          ++cont;

    }

    /* calcular la media y escribir la respuesta */
    media = suma/n;
    printf("\nLa media es %f\n", media);
}
```

Observe que el bucle `while` contiene una instrucción compuesta que, entre otras cosas, se ocupa de incrementar el valor de `cont`. Esto hará que la expresión lógica

```
cont <= n
```

se haga falsa en algún momento, finalizándose con ello el bucle. Observe también que el bucle no se ejecutará ni una sola vez si el valor de `n` es menor que `1` (lo cual por supuesto carecería de sentido).

Supongamos ahora que el programa se utiliza para procesar una lista con los siguientes seis valores: `1, 2, 3, 4, 5, 6`. La ejecución del programa conllevará el siguiente diálogo interactivo. (Observe que las respuestas del usuario están subrayadas.)

```
¿Cuantos números? 6
x = 1
x = 2
x = 3
x = 4
x = 5
x = 6

La media es 3.500000
```

6.4. MÁS SOBRE BUCLES: LA INSTRUCCIÓN `do - while`

Cuando se construye un bucle utilizando la instrucción `while` descrita en la sección 6.3, la comprobación para la continuación del bucle se realiza al *comienzo* de cada pasada. Sin embargo, a veces es deseable disponer de un bucle en el que se realice la comprobación al *final* de cada pasada. Esto se puede hacer mediante la instrucción `do - while`.

La forma general de la instrucción `do - while` es

```
do instrucción while (expresión)
```

La `instrucción` se ejecutará repetidamente, mientras que el valor de `expresión` sea verdadero (es decir, distinto de cero). Observe que `instrucción` siempre se ejecutará al menos una vez, ya que la comprobación de la condición de repetición no se realiza hasta el final de la primera pasada por el bucle. La `instrucción` puede ser simple o compuesta, aunque en la mayoría de las veces será compuesta. Debe incluir algún elemento que altere el valor de `expresión` para que el bucle tenga un final.

Para la mayoría de las aplicaciones es más natural comprobar la condición al comienzo del bucle que al final. Por esta razón, la instrucción `do - while` se utiliza con menor frecuencia que la instrucción `while` descrita en la sección 6.3. Como ilustración del uso de la instrucción `do - while`, repetimos a continuación los ejemplos de programación mostrados en la sección 6.3, utilizando la instrucción `do - while` para los bucles condicionales.

EJEMPLO 6.11. Cantidades enteras consecutivas. En el Ejemplo 6.8 vimos dos programas completos que utilizaban la instrucción `while` para visualizar los números `0, 1, 2, ..., 9`. He aquí otro programa que realiza lo mismo utilizando la instrucción `do - while` en lugar de la instrucción `while`.

```
#include <stdio.h>

main()    /* visualizar los números del 0 al 9 */
{
   int digito = 0;

   do
   printf("%d\n", digito++);
   while (digito <= 9);
}
```

Como en el ejemplo anterior, se le asigna inicialmente a `digito` el valor `0`. El bucle `do - while` visualiza el valor actual de `digito`, incrementando su valor en `1`, y a continuación comprueba si el valor actual de `digito` es mayor que `9`. Si es así, el bucle finaliza; en otro caso, continúa el bucle, utilizando el nuevo valor de `digito`. Observe que la comprobación se lleva a cabo al final de cada pasada por el bucle. El efecto global es que el bucle se repite `10` veces, generándose `10` líneas de salida. Cada línea aparecerá exactamente como se presentó en el Ejemplo 6.8.

Si comparamos este programa con el segundo presentado en el Ejemplo 6.8, veremos que ambos programas tienen aproximadamente el mismo nivel de complejidad. Ninguna de las estructuras de bucle condicionales (`while` o `do - while`) parece mejor que la otra.

EJEMPLO 6.12. Conversión de minúsculas a mayúsculas. Reescribamos ahora el programa presentado en el Ejemplo 6.9, que convertía un texto de minúsculas a mayúsculas, reemplazando los dos bucles `while` por bucles `do - while`. Como en el programa anterior, leeremos primero una línea de texto en minúsculas carácter a carácter, almacenaremos los caracteres en una formación llamada `letras` y lo escribiremos a continuación en mayúsculas utilizando la función de biblioteca `toupper`. Utilizaremos una instrucción `do - while` para leer la línea de texto carácter a carácter y otra instrucción `do - while` para convertir los caracteres a mayúsculas y escribirlos.

He aquí el programa en C.

```
/* convertir una línea de texto de minúsculas a mayúsculas */

#include <stdio.h>
#include <ctype.h>

#define EOL '\n'

main()
{
  char letras[80];
  int aux, cont = -1;

  /* leer el texto en minúsculas */
  do ++cont; while ((letras[cont] = getchar()) != EOL);
  aux = cont;

  /* escribir el texto en mayúsculas */
  cont = 0;
  do  {
          putchar(toupper(letras[cont]));
          ++cont;
  }  while (cont < aux);
}
```

De nuevo vemos dos tipos diferentes de bucles, aunque ambos se escriban como bucles `do - while`. En particular, el número de pasadas por el primer bucle no es conocido *a priori*, pero el segundo bucle ejecuta un número conocido de pasadas, determinado por el valor asignado a `aux`.

Observe que el primer bucle, es decir

```
do ++cont; while ((letras[cont] = getchar()) != EOL);
```

es simple y conciso, pero el segundo bucle,

```
do  {
        putchar(toupper(letras[cont]));
        ++cont;
}  while (cont < aux);
```

es algo más complejo. Ambos bucles se corresponden con los `while` presentados en el Ejemplo 6.9. Observe, sin embargo, que el primer bucle de este programa comienza con un valor de `-1` asignado a `cont`, mientras que el valor inicial de `cont` era 0 en el Ejemplo 6.9.

Cuando se ejecuta el programa, se comporta de la misma forma que el del Ejemplo 6.9.

Antes de dejar este ejemplo, mencionar que el último bucle se podría haber escrito de forma más concisa así:

```
do
    putchar(toupper(letras[cont++]));
while (cont < aux);
```

Esto puede parecer un tanto extraño a los recién iniciados, aunque es característico del estilo de programación comúnmente usado por los programadores de C más experimentados.

EJEMPLO 6.13. Media de una lista de números. Se puede reescribir fácilmente el programa del Ejemplo 6.10 de forma que ilustre el uso de la instrucción `do - while`. La lógica será la misma, excepto que la compobación para determinar si se han introducido los `n` números en la computadora no se hará hasta el final del bucle en lugar de al principio. Por tanto, el programa siempre hará al menos una pasada a través del bucle, aun cuando a `n` se le haya asignado el valor de `0` (lo cual no tendría sentido).

He aquí la versión modificada del programa.

```
/* calcular la media de n números */

#include <stdio.h>

main()
{
   int n, cont = 1;
   float x, media, suma = 0;

   /* inicializar y leer el valor de n */
   printf("¿Cuantos números? ");
   scanf("%d", &n);

   /* leer los números */
   do {
          printf("x = ");
          scanf("%f", &x);
          suma += x;
          ++cont;
   }  while (cont <= n);

   /* calcular la media y escribir la respuesta */
   media = suma/n;
   printf("\nLa media es %f\n", media);
}
```

Cuando se ejecuta el programa, se comporta de la misma forma que la anterior versión presentada en el Ejemplo 6.10.

6.5. MÁS AÚN SOBRE BUCLES: LA INSTRUCCIÓN `for`

La instrucción `for` es la tercera y quizás la más frecuentemente usada de las instrucciones para crear bucles en C. Esta instrucción incluye una expresión que especifica el valor inicial de un índice, otra expresión que determina cuándo se continúa o no el bucle y una tercera expresión que permite que el índice se modifique al final de cada pasada.

La forma general de la instrucción `for` es

```
for (expresión 1; expresión 2; expresión 3) instrucción
```

en donde *`expresión 1`* se utiliza para inicializar algún parámetro (denominado *índice*) que controla la repetición del bucle, *`expresión 2`* representa una condición que debe ser satisfecha

para que se continúe la ejecución del bucle y `expresión 3` se utiliza para modificar el valor del parámetro inicialmente asignado por `expresión 1`. Típicamente, `expresión 1` es una expresión de asignación, `expresión 2` una expresión lógica y `expresión 3` una expresión unaria o una expresión de asignación.

Cuando se ejecuta la instrucción `for`, se evalúa `expresión 2` y se comprueba *antes* de cada pasada por el bucle, y al *final* de cada pasada se evalúa `expresión 3`. Por tanto, la instrucción `for` es equivalente a

```
expresión 1;
while (expresión 2)  {
     instrucción
     expresión 3;
}
```

La ejecución repetida del bucle continuará mientras el valor de `expresión 2` no sea cero, esto es, mientras la condición lógica representada por `expresión 2` sea cierta.

La instrucción `for`, lo mismo que las instrucciones `while` y `do - while`, se puede utilizar para realizar repeticiones donde no se conozca *a priori* el número de pasadas por el bucle. Sin embargo, debido a los elementos incorporados en la instrucción `for`, ésta es particularmente adecuada para bucles donde se conozca *a priori* el número de pasadas. Como regla práctica, los bucles `while` se utilizan generalmente cuando *no* se conoce *a priori* el número de pasadas, y los bucles `for` se utilizan generalmente cuando *sí* se conoce *a priori* el número de pasadas.

EJEMPLO 6.14. Cantidades enteras consecutivas. Ya hemos visto diferentes versiones de un programa en C que visualiza los números 0, 1, 2, ..., 9, uno en cada línea (ver Ejemplos 6.8 y 6.11). He aquí otro programa que hace lo mismo. Ahora, sin embargo, se utiliza la instrucción `for` en lugar de las instrucciones `while` y `do - while`, como en los ejemplos anteriores.

```
#include <stdio.h>

main()  /* visualizar los números del 0 al 9 */

{
   int digito = 0;

   for (digito = 0; digito <= 9; ++digito)
        printf("%d\n", digito);
}
```

La primera línea de la instrucción `for` contiene tres expresiones, encerradas entre paréntesis. La primera expresión asigna un valor inicial 0 a la variable entera `digito`; la segunda expresión establece que el bucle se repetirá mientras que el valor actual de `digito` no sea mayor que 9 al *comienzo* de cada pasada; y la tercera expresión incrementa en `1` el valor de `digito` en cada pasada por el bucle. La función `printf` incluida en el bucle `for` produce la salida deseada, como se vio en el Ejemplo 6.8.

Desde el punto de vista sintáctico no es necesario que se encuentren presentes las tres expresiones en la instrucción `for`, aunque deben aparecer los puntos y comas. Sin embargo, se deben

entender claramente las consecuencias de esta omisión. Se pueden omitir las expresiones primera y tercera si se inicializa y/o altera el índice de alguna otra forma. Sin embargo, si se omite la segunda expresión, se asumirá que ésta tiene un valor permanente de `1` (cierta); por tanto, el bucle continuará ejecutándose repetidamente de forma indefinida a menos que se finalice mediante algún otro mecanismo, tal como una instrucción `break` o `return` (ver secciones 6.8 y 7.2). En cualquier caso, la mayoría de las aplicaciones de la instrucción `for` incluyen las tres expresiones.

EJEMPLO 6.15. Variación de cantidades enteras consecutivas. Veamos otro ejemplo de un programa en C que genera los enteros consecutivos `0`, `1`, `2`,..., `9`, cada uno de ellos en una línea. Utilizaremos ahora una instrucción `for` en la que dos de las tres expresiones se han omitido.

```
#include <stdio.h>

main()   /* visualizar los números del 0 al 9 */

{
   int digito = 0;

   for (; digito <= 9; )
        printf("%d\n", digito++);
}
```

Esta versión del programa es más oscura que la que se mostró en el Ejemplo 6.14, y por tanto menos deseable.

Nótese la similitud entre este programa y el segundo programa del Ejemplo 6.8, que utiliza la instrucción `while`.

EJEMPLO 6.16. Conversión de minúsculas a mayúsculas. Presentamos de nuevo un programa en C que convierte un texto de minúsculas a mayúsculas. Ya hemos visto otros programas que hacen esto, en los Ejemplos 6.9 y 6.12. En éste haremos uso de la instrucción `for` en lugar de las instrucciones `while` y `do - while`.

Como en los casos anteriores, primero se leerá una línea de texto en minúsculas carácter a carácter, se almacenarán los caracteres en un array llamada `letras` y se escribirá a continuación en mayúsculas utilizando la función de biblioteca `toupper`. Se necesitan dos bucles: uno para leer y almacenar los caracteres en minúsculas, el otro para visualizar los caracteres en mayúsculas. Observe que se hace uso de una instrucción `for` para construir un bucle en el que el número de pasadas no es conocido *a priori*.

He aquí el programa completo en C.

```
/* convertir una línea de texto de minúsculas a mayúsculas */

#include <stdio.h>
#include <ctype.h>

#define EOL '\n'

main()

{
   char letras[80];
   int aux, cont;
```

```
    /* leer el texto en minúsculas */
    for(cont = 0; (letras[cont] = getchar()) != EOL; ++cont)
    ;
    aux = cont;

    /* escribir el texto en mayúsculas */
    for (cont = 0; cont < aux; ++cont)
        putchar(toupper(letras[cont]));
}
```

Si se compara este programa con los correspondientes de los Ejemplos 6.9 y 6.12, se puede apreciar que los bucles son más concisos al usar la instrucción `for` que con las instrucciones `while` o `do - while`.

EJEMPLO 6.17. Media de una lista de números. Modifiquemos ahora el programa dado en el Ejemplo 6.10, que calcula la media de una lista de `n` números, de forma que el bucle se realice mediante una instrucción `for`. La lógica será esencialmente la misma, aunque algunos de los pasos se realizarán en orden ligeramente diferente. En particular:

1. Asignar el valor 0 a la variable en coma flotante `suma`.
2. Leer el valor de la variable entera `n`.
3. Asignar el valor `1` a la variable entera `cont`, en donde `cont` es un índice que cuenta el número de pasadas por el bucle.
4. Realizar los siguientes pasos repetidamente, mientras que el valor de `cont` no sea mayor que `n`.
   a) Leer uno de los números de la lista. Cada número se almacenará en la variable en coma flotante `x`.
   b) Añadir el valor de `x` al valor actual de `suma`.
   c) Incrementar en `1` el valor de `cont`.
5. Dividir el valor de `suma` por `n` para obtener la media deseada.
6. Escribir el valor calculado de la media.

Presentamos a continuación el programa en C. Observe que los pasos 3 y 4 se encuentran combinados en la instrucción `for`, y que los pasos 3 y 4(*c*) se realizan ambos en la primera línea (expresiones primera y tercera, respectivamente). Observe también que las operaciones de entrada se encuentran todas acompañadas de mensajes que piden al usuario la información deseada.

```
/* calcular la media de n números */

#include <stdio.h>

main()

{
    int n, cont = 1;
    float x, media, suma = 0;

    /* inicializar y leer el valor de n */
    printf("¿Cuantos números? ");
    scanf("%d", &n);
```

```
    /* leer los números */
    for (cont = 1; cont <= n; ++cont)  {
        printf("x = ");
        scanf("%f", &x);
        suma += x;
    }

    /* calcular la media y escribir la respuesta */
    media = suma/n;
    printf("\nLa media es %f\n", media);
}
```

Comparando este programa con los correspondientes presentados en los Ejemplos 6.10 y 6.13, vemos de nuevo una mayor concisión en el bucle cuando se utiliza la instrucción `for` en lugar de las instrucciones `while` o `do - while`. Sin embargo, ahora la instrucción `for` es en cierto modo más complicada que en los ejemplos anteriores. En particular, observe que la parte *instrucción* del bucle es ahora una instrucción compuesta. Además, debemos asignar de forma explícita un valor inicial a `suma` antes de la instrucción `for`.

Cuando se ejecuta el programa, genera la misma salida que las anteriores versiones de los Ejemplos 6.10 y 6.13.

6.6. ESTRUCTURAS DE CONTROL ANIDADAS

Los bucles, al igual que las instrucciones `if - else`, se pueden anidar, uno dentro de otro. Los bucles interno y externo no necesitan ser generados por el mismo tipo de estructura de control. Sin embargo, es esencial que un bucle se encuentre completamente incluido dentro del otro. Además, cada bucle debe ser controlado por un índice diferente.

Es más, las estructuras de control anidadas pueden involucrar tanto bucles como instrucciones `if - else`. De este modo, un bucle puede estar incluido dentro de una instrucción `if - else`, y una instrucción `if - else` puede estar incluida en un bucle. Las estructuras anidadas pueden ser todo lo complejas que se requiera, adaptándose a la lógica del programa.

EJEMPLO 6.18. Repetición de la media de una lista de números. Supongamos que queremos calcular la media de varias listas consecutivas de números. Si conocemos por adelantado el número de listas de las que vamos a calcular las medias, podemos usar una instrucción `for` para controlar el número de veces que el bucle interno (el cual calcula la media) se va a ejecutar. El cálculo de la media se puede realizar utilizando cualquiera de los tres métodos presentados anteriormente en los Ejemplos 6.10, 6.13 y 6.17 (utilizando una instrucción `while`, `do - while` o `for`).

Tomemos, por ejemplo, la instrucción `for` para calcular la media, como en el Ejemplo 6.17. Procederemos, por tanto, de la siguiente forma:

1. Leer el valor de `nlistas`, una cantidad entera que indica el número de listas que se van a procesar.
2. Leer repetidamente una lista de números y determinar su media. Esto es, calcular la media de una lista de números para cada valor sucesivo de `contlista`, desde `1` hasta `nlistas`. Seguir los pasos dados en el Ejemplo 6.14 para calcular cada media.

He aquí el programa en C.

```
/* calcular la media de varias listas de números */

#include <stdio.h>

main()

{
   int n, cont, nlistas, contlista;
   float x, media, suma;

   /* leer el número de listas */
   printf("¿Cuantas listas? ");
   scanf("%d", &nlistas);

   /* bucle externo (procesar cada lista de números) */
   for (contlista = 1; contlista <= nlistas; ++contlista)  {

       /* inicializar y leer el valor de n */
       suma = 0;
       printf("\nLista número %d\n¿Cuantos números? ", contlista);
       scanf("%d", &n);

       /* leer los números */
       for (cont = 1; cont <= n; ++cont)  {
            printf("x = ");
            scanf("%f", &x);
            suma += x;
       }     /* fin del bucle interno */

       /* calcular la media y escribir la respuesta */
       media = suma/n;
       printf("\nLa media es %f\n", media);

   }    /* fin del bucle externo */
}
```

Este programa contiene varios elementos interesantes. Primero, contiene dos instrucciones `for`, una dentro de otra. Cada instrucción `for` incluye una instrucción compuesta, que consta de varias instrucciones individuales encerradas entre llaves. Se utiliza un índice diferente en cada instrucción `for` (los índices son `contlista` y `cont`, respectivamente).

Observe que `suma` se debe inicializar ahora dentro del bucle externo y no en la declaración. Esto permite que `suma` sea puesta a cero cada vez que se introduce un nuevo conjunto de datos (es decir, al comienzo de cada pasada a través del bucle externo).

Las operaciones de entrada se encuentran todas acompañadas de la presentación de rótulos y mensajes, indicando al usuario qué datos son requeridos. De esta forma, vemos pares de funciones `printf` y `scanf` en varios lugares a lo largo del programa. Dos de las funciones `printf` contienen varios caracteres de `nueva línea`, para controlar el espaciamiento de las líneas de salida. Esto hace que la salida asociada a cada conjunto de datos (cada pasada por el bucle externo) se identifique con facilidad.

Finalmente, observe que el programa está organizado en segmentos separados fácilmente identificables, con cada segmento precedido por una línea en blanco y un comentario.

Cuando se ejecuta el programa introduciendo tres conjuntos sencillos de datos, se genera el siguiente diálogo. Como de costumbre, las respuestas del usuario a los rótulos se encuentran subrayadas.

```
¿Cuantas listas? 3

Lista número 1
¿Cuantos números? 4
x = 1.5
x = 2.5
x = 6.2
x = 3.0

La media es 3.300000

Lista número 2
¿Cuantos números? 3
x = 4
x = -2
x = 7

La media es 3.000000

Lista número 3
¿Cuantos números? 5
x = 5.4
x = 8.0
x = 2.2
x = 1.7
x = -3.9

La media es 2.680000
```

EJEMPLO 6.19. Conversión de varias líneas de texto a mayúsculas. Este ejemplo ilustra la utilización de dos tipos distintos de bucles, uno anidado dentro del otro. Ampliaremos los programas de conversión de minúsculas a mayúsculas presentados en los Ejemplos 6.9, 6.12 y 6.16 para que varias líneas de texto en minúsculas se conviertan a mayúsculas, realizándose la conversión línea a línea. En otras palabras, leeremos una línea de texto en minúsculas, la visualizaremos en mayúsculas y a continuación procesaremos otra línea, y así sucesivamente. Esto continuará hasta que se detecte una línea cuyo primer carácter sea un asterisco.

Utilizaremos bucles anidados para llevar a cabo el proceso. El bucle externo se utilizará para procesar varias líneas de texto. Éste incluirá dos bucles internos separados. El primero de ellos lee una línea de texto y el segundo visualiza el texto convertido en mayúsculas. Observe que estos dos bucles internos no están anidados entre sí. Utilicemos, por ejemplo, una instrucción `while` para el bucle externo y una instrucción `for` para cada uno de los bucles internos.

En términos generales, el proceso será el siguiente:

1. Realizar repetidamente cada uno de los siguientes pasos, para cada línea de texto, mientras el primer carácter de la línea no sea un asterisco.

   a) Leer una línea de texto y asignar cada carácter a cada elemento de un array de caracteres `letras`. Se definirá la línea como una sucesión de caracteres terminados por una marca fin de línea (nueva línea).

b) Asignar la cuenta de caracteres (incluyendo el carácter de fin de línea) a `aux`.

c) Escribir la línea en mayúsculas, utilizando la función de biblioteca `toupper` para realizar la conversión. Escribir a continuación dos caracteres de nueva línea de forma que la siguiente línea de entrada se encuentre separada de la salida actual por una línea en blanco.

2. Una vez que se ha detectado un asterisco como primer carácter de una nueva línea, escribir `Hasta_luego` y terminar el proceso.

He aquí el programa en C completo.

```
/*  convertir varias líneas de texto de minúsculas
    a mayúsculas. Continuar la conversión hasta que
    primer carácter de una línea sea un asterisco (*)   */

#include <stdio.h>
#include <ctype.h>

#define EOL '\n'

main()
{
   char letras[80];
   int aux, cont;

   while ((letras[0] = getchar()) != '*')     {

          /* leer línea de texto */
          for(cont = 0; (letras[cont] = getchar()) != EOL; ++cont)
              ;
          aux = cont;

          /* escribir la línea de texto */
          for (cont = 0; cont < aux; ++cont)
               putchar(toupper(letras[cont]));
          printf("\n\n");
   }      /* fin del bucle externo */

   printf("Hasta luego");
}
```

A continuación se muestra una sesión interactiva típica, para ilustrar la ejecución del programa. Observe que el texto de entrada proporcionado por el usuario se encuentra subrayado, como de costumbre.

```
Mil gracias derramando paso por estos sotos con presura . . .
MIL GRACIAS DERRAMANDO PASO POR ESTOS SOTOS CON PRESURA . . .

En un lugar de la Mancha de cuyo nombre no quiero acordarme . . .
EN UN LUGAR DE LA MANCHA DE CUYO NOMBRE NO QUIERO ACORDARME . . .

*
Hasta luego
```

Debe quedar claro que la decisión de utilizar una instrucción `while` para el bucle externo e instrucciones `for` para los bucles internos es arbitraria. Se podrían haber utilizado otras estructuras para los bucles.

Muchos programas involucran tanto repetición como ejecución condicional. Frecuentemente se anidan diferentes estructuras de control, unas dentro de otras, como mostramos en los tres ejemplos siguientes.

EJEMPLO 6.20. Codificación de una cadena de caracteres. Escribamos un programa sencillo en C que lea una secuencia de caracteres ASCII y escriba en su lugar una secuencia de caracteres codificados. Si un carácter es una letra o dígito, lo reemplazaremos por el siguiente carácter en el conjunto de caracteres, excepto `Z` que será reemplazado por `A`, `z` por `a` y `9` por `0`. Por tanto, `1` se transforma en `2`, `C` en `D`, `p` en `q`, etc. Cualquier carácter que no sea letra o dígito se reemplazará por un punto (.).

El proceso comenzará por la lectura de los caracteres. Se utilizará la función `scanf` para este fin. Se introducirán y almacenarán todos los caracteres hasta el carácter de nueva línea (`\n`), pero sin incluir éste, en un array de 80 elementos de tipo carácter llamado `linea`.

A continuación se codificarán y visualizarán los caracteres individualmente dentro de un bucle `for`. El bucle procesará cada uno de los caracteres de `linea` hasta encontrar el carácter de escape `\0`, que indica el final de la secuencia de caracteres. (Recuérdese que la secuencia de escape `\0` se añade automáticamente al final de cada cadena de caracteres.) Dentro del bucle se incluyen varias instrucciones `if - else` anidadas para realizar la codificación adecuada. Se visualizará cada carácter codificado utilizando la función `putchar`.

A continuación se muestra el programa en C completo.

```
/* leer una cadena de caracteres, reemplazar a continuación
   cada carácter por un carácter codificado correspondiente */

#include <stdio.h>

main()

{
   char linea[80];
   int cont;

   /* leer la cadena completa */

   printf("Introducir debajo una línea de texto:\n");
   scanf("%[^\n]", linea);

   /* codificar cada carácter y escribirlo */

   for (cont = 0; linea[cont] != '\0'; ++cont)  {
       if (((linea[cont] >= '0') && (linea[cont] < '9')) ||
          ((linea[cont] >= 'A') && (linea[cont] < 'Z')) ||
          ((linea[cont] >= 'a') && (linea[cont] < 'z')))
                 putchar(linea[cont] + 1);
       else if (linea[cont] == '9') putchar('0');
            else if (linea[cont] == 'Z') putchar('A');
                 else if (linea[cont] == 'z') putchar('a');
                      else putchar('.');
   }
}
```

La ejecución de este programa genera el siguiente diálogo a modo de ilustración. Se ha subrayado de nuevo la entrada proporcionada por el usuario.

```
Introducir debajo una línea de texto:
The White House, 1600 Pennsylvania Avenue, Washington, DC
Uif.Xijuf.Ipvtf..2711.Qfootzmwbojb.Bwfovf..Xbtijohupo..ED
```

EJEMPLO 6.21. Cálculos repetidos del interés compuesto con detección de error. En el Ejemplo 5.2 vimos un programa en C que realizaba cálculos simples del interés compuesto, como se indicaba en el Ejemplo 5.1. Sin embargo, este programa no permitía la ejecución repetida (varios cálculos sucesivos, utilizando diferentes conjuntos de datos de entrada para cada uno de ellos), ni contemplaba la detección de errores en los datos de entrada. Nos ocuparemos a continuación de añadir estos elementos al programa anterior.

Incluiremos los cálculos anteriores en una instrucción `while`, que se ejecutará mientras el valor introducido de la suma inicial (`P`) sea positivo. Por tanto, la introducción del valor cero para `P` se interpretará como condición de parada. Incluiremos un mensaje en el que se explique la forma de parar cuando se pida el valor de `P`.

Además, incluiremos una *detección de errores* que compruebe si el valor de cada entrada es negativo, ya que un valor negativo no tiene sentido y se interpretará como un error. Cada comprobación se realizará en una instrucción `if`. Si se detecta un error (un valor negativo), se presentará un mensaje pidiendo al usuario que introduzca de nuevo el dato.

He aquí el programa en C.

```
/* problema sencillo de interés compuesto */

#include <stdio.h>
#include <math.h>

main()

{
   float p, r, n, i, f;

   /* lee el valor de la suma inicial */

   printf("Por favor, introduce la suma inicial (P) ");
   printf("\n(Para finalizar el programa, introducir 0 como valor): ");
   scanf("%f", &p);
   if (p < 0)  {
      printf("\nERROR - inténtelo de nuevo, por favor: ");
      scanf("%f", &p);
   }
   while (p > 0)  {    /* bucle principal */

      /* leer los restantes datos de entrada */

      printf("\nPor favor, introduce el interés (r): ");
      scanf("%f", &r);
```

```
      if (r < 0) {
         printf("\nERROR - Inténtelo de nuevo, por favor: ");
         scanf("%f", &r);
      }
      printf("\nPor favor, introduce el número de años (n): ");
      scanf("%f", &n);
      if (n < 0)  {
         printf("\nERROR - Inténtelo de nuevo, por favor: ");
         scanf("%f", &n);
      }

      /* calcular i y f */

      i = r/100;
      f = p * pow((1 + i), n);

      /* escribir salida */

      printf("\nEl valor final (F) es: %.2f\n", f);

      /* leer la suma inicial para la siguientes pasada */

      printf("\n\nPor favor, introduce la suma inicial (P) ");
      printf("\n(Para finalizar el programa, introducir 0 como valor): ");
      scanf("%f", &p);
      if (p < 0)  {
         printf("\nERROR - inténtelo de nuevo, por favor: ");
         scanf("%f", &p);
      }
   }   /* fin del bucle principal */
}
```

A continuación se muestra una sesión interactiva típica. Observe de nuevo que se han subrayado las respuestas del usuario.

```
Por favor, introduce la suma inicial (P)
(Para finalizar el programa, introducir 0 como valor): 1000

Por favor, introduce el interés (r): 6

Por favor, introduce el número de años (n): 20

El valor final (F) es: 3207.14


Por favor, introduce la suma inicial (P)
(Para finalizar el programa, introducir 0 como valor): 5000

Por favor, introduce el interés (r): -7.5

ERROR - Inténtalo de nuevo, por favor: 7.5
```

```
Por favor, introduce el número de años (n): 12

El valor final (F) es: 11908.90


Por favor, introduce la suma inicial (P)
(Para finalizar el programa, introducir 0 como valor): 0
```

Observe que se han proporcionado dos conjuntos de datos. El primer conjunto es completamente correcto, resultando un valor futuro de 3207.14 (como en el Ejemplo 5.4). En el segundo conjunto de datos se introduce inicialmente un valor negativo del tanto por ciento de interés (r). Esto es detectado como error, presentando a continuación un mensaje de error y solicitando otro valor. Una vez que se ha proporcionado el valor correcto, el resto de la ejecución continúa como era de esperar.

Después de que se haya procesado el segundo conjunto de datos, el usuario introduce el valor 0 para la suma inicial en respuesta al mensaje. Esto hace que termine la ejecución del programa.

Debe quedar claro que la detección de errores utilizada de este programa se limita a la introducción de cantidades en coma flotante negativas como datos de entrada. Otro tipo de error ocurriría si se introdujese una letra o un signo de puntuación en una de las cantidades de entrada requeridas. Esto producirá un error de entrada en la función scanf. Cada compilador trata este error de forma diferente, impidiendo una detección de errores general y sencilla.

El siguiente programa es de naturaleza más comprensible. Incluye la mayoría de los elementos de programación que hemos tratado con anterioridad en este libro.

EJEMPLO 6.22. Solución de una ecuación algebraica. Para los lectores más aficionados a las matemáticas, este ejemplo ilustra cómo se pueden utilizar las computadoras para resolver ecuaciones algebraicas, incluidas aquellas que no se pueden resolver utilizando métodos más directos. Consideremos, por ejemplo, la ecuación

$$x^5 + 3x^2 - 10 = 0.$$

Esta ecuación no se puede transformar de forma que obtengamos una solución exacta para x. Sin embargo, se puede determinar la solución mediante un procedimiento repetido de prueba y error (procedimiento *iterativo*) que refina sucesivamente un valor inicial supuesto.

Comenzaremos por reescribir la ecuación de la forma siguiente:

$$x = (10 - 3x^2)^{1/5}$$

Nuestro procedimiento empezará por suponer un valor de x, sustituir este valor en la parte derecha de la última ecuación y a continuación calcular un nuevo valor de x. Si este valor nuevo es igual (o muy próximo) al valor anterior, es que hemos obtenido la solución de la ecuación. De otra forma se sustituirá este nuevo valor en la parte derecha de la ecuación y se volverá a obtener otro valor de x, y así sucesivamente. Continuará este procedimiento hasta que los valores sucesivos de x sean lo suficientemente próximos (esto es, hasta que el cómputo *converja*) o hasta que se exceda un número especificado de iteraciones. La última condición se ocupa de impedir que continúen los cálculos indefinidamente en el caso de que los resultados obtenidos no converjan.

Para ver cómo funciona el método, supongamos un valor inicial de x = 1.0. Sustituyendo este valor en la parte derecha de la ecuación obtenemos:

$$x = [10 - 3(1.0)^2]^{0.2} = 1.47577$$

A continuación sustituimos este nuevo valor en la ecuación, obteniendo

$$x = [10 - 3(1.47577)^2]^{0.2} = 1.28225$$

Continuando este procedimiento, se obtiene

$$x = [10 - 3(1.28225)^2]^{0.2} = 1.38344$$

$$x = [10 - 3(1.38344)^2]^{0.2} = 1.33613$$

y así sucesivamente. Observe que los valores consecutivos de `x` parecen converger a la respuesta final.

El éxito del método depende del valor inicial tomado para `x`. Si este valor es demasiado grande en valor absoluto, la cantidad entre corchetes será negativa, y no se puede elevar una cantidad negativa a un exponente fraccionario. Por tanto, debemos comprobar si el valor de `10 - 3x²` es negativo antes de sustituir el valor de `x` por el de la parte derecha de la ecuación.

Con vistas a la escritura del programa, definamos los siguientes símbolos:

`cont`	=	un contador de iteraciones (`cont` se incrementará en 1 en cada iteración)
`valor`	=	el valor de `x` sustituido en la parte derecha de la ecuación
`raiz`	=	el nuevo valor calculado de `x`
`test`	=	la cantidad (`10 - 3`x^2)
`error`	=	el valor absoluto de la diferencia entre `raiz` y `valor`
`indicador`	=	una variable entera que indica la continuación o no de la iteración

Continuaremos los cálculos hasta que se satisfaga una de las siguientes condiciones:

1. El valor de `error` es menor que `0.00001`, en cuyo caso habremos obtenido una solución satisfactoria.
2. Se han realizado cincuenta iteraciones (`cont = 50`).
3. La variable `test` tiene valor negativo, en cuyo caso no pueden continuar los cálculos.

Controlemos el desarrollo de los cálculos escribiendo cada valor sucesivo de `raiz`.

Ahora podemos escribir el siguiente esquema del programa.

1. Por conveniencia, definir las constantes simbólicas `VERDADERO` y `FALSO`.
2. Declarar todas las variables e inicializar las variables enteras `indicador` y `cont` (asignar `VERDADERO` a `indicador` y `0` a `cont`).
3. Leer el valor inicial de `valor`.
4. Llevar a cabo el siguiente procedimiento iterativo mientras `indicador` sea `VERDADERO`.
   a) Incrementar el valor de `cont` en `1`.
   b) Asignar `FALSO` a `indicador` si el nuevo valor de `cont` es igual a `50`. Esto indica la última pasada por el bucle.
   c) Examinar el valor de `test`. Si su valor es positivo, proceder como sigue.
      i) Calcular un nuevo valor para `raiz`; escribir el valor actual de `cont`, seguido del valor actual de `raiz`.
      ii) Evaluar `error`, que es el valor absoluto de la diferencia entre `raiz` y `valor`. Si este valor es mayor que `0.00001`, asignar el valor actual de `raiz` a `valor` y continuar con las siguiente iteración. De otra forma, escribir los valores actuales de `raiz` y `cont`, y poner `indicador` a `FALSO`. El valor actual de `raiz` se considerará la solución deseada.

d) Si el valor actual de `test` no es positivo, no se puede seguir con los cálculos. Por tanto, escribir un mensaje de error apropiado (por ejemplo `Números fuera de rango`) y establecer `indicador` a `FALSO`.

5. Después de completar el paso 4, escribir un mensaje de error adecuado (por ejemplo `Convergencia no obtenida`) si `cont` tiene el valor de `50` y el valor de `error` es mayor que `0.00001`.

Expresemos ahora el guión del programa en la forma de pseudocódigo con vistas a simplificar la transformación del esquema a programa en C.

```
#include archivos

#define constantes simbólicas

main()

{
  /* declaración e inicialización de variables */

  /* leer parámetros de entrada */

  while (indicador)  {

        /* incrementar cont */

        /* asignar a indicador FALSO si cont = 50 */

        /* evaluar test */

        if (test > 0)  {
            /* evaluar raiz */
            /* visualizar cont y raiz */
            /* evaluar error */

            if (error > 0.00001) valor = raiz;
            else  {
                /* asignar a indicador FALSO */
                /* visualizar la respuesta final (raiz y cont) */
            }
        }

        else  {
            /* asignar a indicador FALSO */
            /* números fuera de rango - escribir mensaje de error */
        }

  }  /* fin de while */

  if ((cont == 50) && (error > 0.00001))
     /* convergencia no obtenida - escribir mensaje de error */
}
```

He aquí el programa en C completo.

```
/* determinar las raíces de una ecuación algebraica
   utilizando un procedimiento iterativo */

#include <stdio.h>
#include <math.h>

#define VERDADERO 1
#define FALSO 0

main()

{
   int indicador = VERDADERO, cont = 0;
   float valor, raiz, test, error;

   /* leer parámetros de entrada */

   printf("Valor inicial: ");
   scanf("%f", &valor);
   while (indicador)   {          /* comienza bucle principal */
          ++cont;
          if (cont == 50) indicador = FALSO;
          test = 10. - 3. * valor * valor;
          if (test > 0)   {       /* otra iteración */
             raiz = pow(test, 0.2);
             printf("\nIteración número: %2d", cont);
             printf("    x= %7.5f", raiz);
             error = fabs(raiz - valor);
             if (error > 0.00001) valor = raiz; /* repetir el cálculo */
             else   {             /* visualizar la respuesta final */
                indicador = FALSO;
                printf("\n\nRaíz= %7.5f", raiz);
                printf("    Nº de iteraciones = %2d", cont);
             }
          }
          else  {                 /* mensaje de error */
             indicador = FALSO; ");
             printf("\nNúmeros fuera de rango -");
             printf ("intenta con otro valor  inicial");
          }
     }
     if ((cont == 50) && (error > 0.00001)) /* otro mensaje de error */
        printf("\n\nConvergencia no obtenida tras 50 iteraciones");
}
```

Observe que el programa contiene una instrucción `while` y varias instrucciones `if - else`. Se podría haber utilizado fácilmente una instrucción `for` en lugar de la instrucción `while`. Observe también la existencia de instrucciones `if - else` anidadas en la mitad del programa.

A continuación se muestra la salida que se genera para un valor inicial de `x = 1`, con la respuesta del usuario subrayada. Nótese que los cálculos han convergido a la solución `x = 1.35195` después de `16` iteraciones. La salida muestra los sucesivos valores de `x` acercándose más y más, conduciendo a la solución final.

```
Valor inicial: 1

Iteración número:  1     x= 1.47577
Iteración número:  2     x= 1.28225
Iteración número:  3     x= 1.38344
Iteración número:  4     x= 1.33613
Iteración número:  5     x= 1.35951
Iteración número:  6     x= 1.34826
Iteración número:  7     x= 1.35375
Iteración número:  8     x= 1.35109
Iteración número:  9     x= 1.35238
Iteración número: 10     x= 1.35175
Iteración número: 11     x= 1.35206
Iteración número: 12     x= 1.35191
Iteración número: 13     x= 1.35198
Iteración número: 14     x= 1.35196
Iteración número: 15     x= 1.35196
Iteración número: 16     x= 1.35195

Raíz = 1.35195    Nº de iteraciones = 16
```

Supongamos ahora que se introduce como valor inicial `x = 10`. Este valor genera un número de `test` negativo en la primera iteración. Por tanto, la salida tiene el aspecto siguiente.

```
Valor inicial: 10
Números fuera de rango - intenta con otro valor inicial
```

Es interesante ver qué ocurre cuando se vuelve a tomar como valor inicial `x = 1`, pero el máximo número de iteraciones cambia de `50` a `10`. Intente esto y observe el resultado.

Hay que decir que existen otros muchos métodos iterativos para la resolución de ecuaciones algebraicas. La mayoría convergen de forma más rápida que el método descrito anteriormente (requieren menos iteraciones para conseguir una solución), aunque implican mayor complejidad matemática.

6.7. LA INSTRUCCIÓN `switch`

La instrucción `switch` hace que se seleccione un grupo de instrucciones entre varios grupos disponibles. La selección se basa en el valor de una expresión que se incluye en la instrucción `switch`.

La forma general de la instrucción `switch` es

```
switch (expresión) instrucción
```

en donde *expresión* devuelve un valor entero. Tenga en cuenta que *expresión* también puede ser de tipo `char`, ya que los caracteres individuales tienen valores enteros.

La *instrucción* incluida es generalmente una instrucción compuesta que especifica opciones posibles a seguir. Cada opción se expresa como un grupo de una o más instrucciones individuales dentro de la *instrucción* global incluida.

Para cada opción, la primera instrucción dentro del grupo debe ser precedida por una o más *etiquetas «case»* (también llamadas *prefijos*). Las etiquetas `case` identifican los diferentes grupos de instrucciones (las distintas opciones) y distinguen unas de otras. Las etiquetas `case` deben ser, por tanto, únicas dentro de una instrucción `switch` dada.

Cada grupo de instrucciones se escribe de forma general:

```
case expresión :
     instrucción 1
     instrucción 2
     . . . . .
     instrucción n
```

o en el caso de varias etiquetas `case`,

```
case expresión 1 :
case expresión 2 :
    . . . . .
case expresión m :
     instrucción 1
     instrucción 2
     . . . . .
     instrucción n
```

en donde *expresión 1*, *expresión 2*, ..., *expresión m* representan expresiones constantes de valores enteros. Normalmente, cada una de estas expresiones se escribirá o bien como constante entera o como constante de carácter. Cada *instrucción* individual que sigue a las etiquetas `case` puede ser simple o compuesta.

Cuando se ejecuta la instrucción `switch`, se evalúa la *expresión* y se transfiere el control directamente al grupo de instrucciones cuya etiqueta `case` tenga el mismo valor que el de *expresión*. Si ninguno de los valores de las etiquetas `case` coincide con el valor de *expresión*, entonces no se seleccionará ninguno de los grupos de la instrucción `switch`. En este caso se transfiere el control directamente a la instrucción que se encuentre a continuación de la instrucción `switch`.

EJEMPLO 6.23. Presentamos a continuación una instrucción `switch` sencilla. En este ejemplo, suponemos que `eleccion` es una variable de tipo `char`.

```
switch (eleccion = getchar())  {

case 'r':
case 'R':
     printf("ROJO");
     break;
```

```
case 'b':
case 'B':
     printf("BLANCO");
     break;

case 'a':
case 'A':
     printf("AZUL");
}
```

Por tanto, se presentará `ROJO` si `eleccion` representa `r` o `R`, se presentará `BLANCO` si `eleccion` representa `b` o `B`, y `AZUL` si `eleccion` tiene el valor `a` o `A`. No se visualizará nada si `eleccion` tiene asignado algún otro carácter.

Observe que cada grupo de instrucciones tiene dos etiquetas `case` para contemplar mayúsculas y minúsculas. Observe también que cada uno de los dos primeros grupos acaba con la instrucción `break` (ver sección 6.8). La instrucción `break` hace que se transfiera el control fuera de la instrucción `switch`, evitando que se ejecute más de un grupo de instrucciones.

Uno de los grupos de instrucciones se puede etiquetar como `default`. Este grupo se seleccionará si ninguna de las etiquetas `case` coincide con el valor de *`expresión`*. (Ésta es una forma conveniente de generar un mensaje de error en rutinas de corrección de errores.) El grupo `default` puede aparecer en cualquier lugar dentro de la instrucción `switch` (no necesita ser emplazado al final). Si ninguna de las etiquetas `case` coincide con el valor de *`expresión`* y no se encuentra el grupo `default` (como en el ejemplo anterior), la instrucción `switch` no hará nada.

EJEMPLO 6.24. He aquí una variación de la instrucción `switch` presentada en el Ejemplo 6.23.

```
switch (eleccion = toupper(getchar()))  {

case 'R':
     printf("ROJO");
     break;

case 'B':
     printf("BLANCO");
     break;

case 'A':
     printf("AZUL");
     break;

default:
     printf("ERROR");
}
```

La instrucción `switch` contiene ahora un grupo `default` (que consta de una sola instrucción), el cual genera un mensaje de error si ninguna de las etiquetas `case` coincide con *`expresión`*.

Cada uno de los tres primeros grupos de instrucciones tiene ahora una sola etiqueta `case`. En este ejemplo se necesitan etiquetas `case` múltiples, ya que la función de biblioteca `toupper` hace que todos los caracteres que se reciban se conviertan a mayúsculas. Por tanto, `eleccion` siempre tendrá asignada una letra mayúscula.

EJEMPLO 6.25. He aquí otra típica instrucción `switch`. En este ejemplo, se supone que `indicador` es una variable entera, y se supone que `x` e `y` son variables en coma flotante.

```
switch (indicador)   {

case -1:
     y = fabs(x);
     break;

case 0:
     y = sqrt(x);
     break;

case 1:
     y = x;
     break;

case 2:
case 3:
     y = 2 * (x - 1);
     break;

default:
     y = 0;
}
```

En este ejemplo se le asignará a `y` algún valor relacionado con `x` si `indicador` es igual a `-1, 0, 1, 2` o `3`. La relación exacta entre `y` y `x` dependerá del valor particular de `indicador`. Si `indicador` representa algún otro valor, entonces se le asignará a `y` el valor `0`.

Observe que las etiquetas `case` son numéricas en este ejemplo. Observe también que el cuarto grupo de instrucciones tiene dos etiquetas `case`, mientras que cada uno de los otros grupos sólo tiene una etiqueta `case`. Y finalmente observe que en esta instrucción `switch` se incluye un grupo por omisión (que consta de una sola instrucción).

De forma práctica, la instrucción `switch` se puede ver como una alternativa al uso de instrucciones `if - else` que comprueben igualdades. En tales situaciones, utilizar la instrucción `switch` es por norma general mucho más conveniente.

EJEMPLO 6.26. Cálculo de la depreciación. Consideremos cómo calcular la depreciación anual para algunos objetos susceptibles a ello, tales como un edificio, una máquina, etc. Hay tres métodos comúnmente usados para el cálculo de la depreciación, que se suelen llamar método de *línea recta*, método de *balance doblemente declinante*, y el método de *la suma de los dígitos de los años*. Deseamos escribir un programa en C que nos permita seleccionar algunos de estos métodos para cada conjunto de cálculos.

El proceso comenzará leyendo el valor original (sin depreciar) del objeto, la vida del objeto (el número de años en los que se depreciará) y un entero que indique qué método se utilizará. A continuación se calculará la depreciación anual y el valor remanente (no depreciado) del objeto, y se escribirá para cada año.

El método de *línea recta* es el más fácil de utilizar. En este método el valor original del objeto se divide por su vida (número total de años). El cociente resultante será la cantidad en que el objeto se deprecia anualmente. Por ejemplo, si un objeto de 8000 dólares se deprecia en diez años, entonces la depreciación anual será `8000/10 = 800` dólares. Por tanto, el valor del objeto habrá disminuido en 800 dólares cada año. Nótese que la depreciación anual es la misma cada año cuando se utiliza este método.

Cuando se utiliza el método de *balance doblemente declinante*, el valor del objeto disminuye cada año en un *porcentaje* constante. Por tanto, la verdadera cantidad depreciada, en dólares, variará de un año al siguiente. Para obtener el factor de depreciación, dividimos dos por la vida del objeto. Este factor se multiplica por el valor del objeto *al comienzo de cada año* (y no el valor original del objeto) para obtener la depreciación anual.

Supongamos, por ejemplo, que deseamos depreciar un objeto de 8000 dólares en diez años, utilizando el método del balance doblemente declinante. El factor de depreciación será `2/10 = 0.2`. Por tanto, la depreciación el primer año será `0.20 × 8000 = 1600` dólares. La depreciación del segundo año será `0.20 × (8000 - 1600) = 0.20 × 6400 = 1280` dólares; la depreciación del tercer año será `0.20 × 5120 = 1024` dólares, y así sucesivamente.

En el método de *la suma de los dígitos de los años*, el valor del objeto irá disminuyendo en un porcentaje que es *diferente* cada año. El factor de depreciación será una fracción cuyo denominador es la suma de los dígitos de `1` a `n`, en donde `n` representa la vida del objeto. Si, por ejemplo, consideramos un tiempo de vida de diez años, el denominador será `1 + 2 + 3 + ... + 10 = 55`. Para el primer año el numerador será `n`, para el segundo año será `(n - 1)`, para el tercer año `(n - 2)`, y así sucesivamente. La depreciación anual se obtiene multiplicando el factor de depreciación por el valor original del objeto.

Para ver cómo funciona el método de la suma de los dígitos de los años, depreciemos de nuevo un objeto de 8000 dólares en diez años. La depreciación del primer año será `(10/55) × 8000 = 1454.55` dólares; el segundo año será `(9/55) * 8000 = 1309.09` dólares, y así sucesivamente.

Definamos ahora los siguientes símbolos, que nos permitirán escribir el programa.

`val` = el valor en curso del objeto
`aux` = el valor original del objeto (el valor original de `val`)
`deprec` = la depreciación anual
`n` = el número de años por los que se depreciará el objeto
`anual` = un contador que tomará valores de `1` a `n`
`eleccion` = un entero que indica qué método se ha de utilizar

Presentamos a continuación el guión del programa en C.

1. Declarar todas las variables e inicializar la variable entera `eleccion` a `0` (en realidad, podemos asignar cualquier valor distinto de `4` a `eleccion`).
2. Repetir todos los pasos siguientes si el valor de `eleccion` no es igual a `4`.
   a) Leer el valor de `eleccion`, que indica el tipo de cálculo que se llevará a cabo. Este valor sólo puede ser `1`, `2`, `3` o `4`. (Cualquier otro valor se considerará un error.)
   b) Si `eleccion` tiene asignado el valor de `1`, `2` o `3`, leer los valores de `val` y `n`.
   c) Dependiendo del valor asignado a `eleccion`, saltar a la parte adecuada del programa y realizar los cálculos indicados. En particular,
      i) Si `eleccion` tiene asignado el valor de `1`, `2` o `3`, calcular la depreciación anual y el nuevo valor del objeto año a año, utilizando el método apropiado según indique el valor de `eleccion`. Escribir los resultados según se vayan calculando, año a año.

ii) Si `eleccion` tiene asignado el valor 4, escribir el mensaje «Hasta luego» y finalizar el programa saliendo del bucle `while`.
iii) Si `eleccion` tiene asignado un valor distinto de 1, 2, 3 o 4, escribir un mensaje de error e iniciar una nueva pasada por el bucle.

Expresemos ahora este esquema en forma de pseudocódigo.

```
#include archivos

main()
{
  /* declaración e inicialización de variables */

  while (eleccion != 4)  {

        /* generar menú y leer elección */

        if (eleccion >= 1 && eleccion <= 3)
           /* leer val y n */

        switch (eleccion)  {

        case 1:   /* método de línea recta */

             /* escribir título */

             /* calcular depreciación */

             /* para cada año:
                     calcular un nuevo valor
                     escribir año, depreciación y valor */

        case 2:   /* método de balance doblemente declinante */

             /* escribir título */

             /* para cada año:
                     calcular depreciación
                     calcular un nuevo valor
                     escribir año, depreciación y valor */

        case 3:   /* método de la suma de los dígitos de los años */

             /* escribir título */

             /* copiar valor original */

             /* para cada año:
                     calcular depreciación
                     calcular un nuevo valor
                     escribir año, depreciación y valor */
```

```
        case 4:    /* fin de los cálculos */

             /* escribir mensaje "Hasta luego" */

             /* escribir título */

        default:    /* generar mensaje de error */

             /* escribir mensaje de error */
        }
    }
}
```

La mayor parte del pseudocódigo es de fácil comprensión, aunque hay lugar para algunos comentarios. Primero, vemos que se utiliza una instrucción `while` para repetir el conjunto completo de cálculos. Dentro de este bucle global se utiliza una instrucción `switch` para seleccionar el método concreto de cálculo de la depreciación. Cada método utiliza una instrucción `for` para realizar los cálculos requeridos.

Llegados a este punto no es difícil escribir el programa final en C, como se muestra a continuación.

```
/* calcular la depreciación utilizando uno de tres métodos diferentes */

#include <stdio.h>

main()

{
   int n, anual, eleccion = 0;
   float val, aux, deprec;

   while (eleccion != 4)   {

         /* leer datos de entrada */

         printf("\nMétodo: (1-LR  2-BDD  3-SDA  4-Fin) ");
         scanf("%d", &eleccion);
         if (eleccion >= 1 && eleccion <=3)   {
            printf("Valor original: ");
            scanf("%f", &val);
            printf("Número de años: ");
            scanf("%d", &n);
         }

         switch (eleccion)   {

         case 1    /* método de la línea recta */

              printf("\nMétodo de la línea recta\n\n");
              deprec = val/n;
              for (anual = 1; anual <= n; ++anual)   {
                  val -= deprec;
```

```
                printf("Fin de año %2d", anual);
                printf("  Depreciación: %7.2f", deprec);
                printf("  Valor actual: %8.2f\n", val);
            }
            break;
        case 2:    /* método de balance doblemente declinante */

            printf("\nMétodo de balance doblemente declinante\n\n");
            for (anual = 1; anual <= n; ++anual)  {
                deprec = 2*val/n;
                val -= deprec;
                printf("Fin de año %2d", anual);
                printf("  Depreciación: %7.2f", deprec);
                printf("  Valor actual: %8.2f\n", val);
            }
            break;
        case 3:    /* método de la suma de los dígitos de los años */

            printf("\nMétodo de la suma de los dígitos");
            printf ("de los años\n\n");
            aux = val;
            for(anual = 1; anual <= n; ++anual)  {
                deprec = (n-anual+1)*aux / (n*(n+1)/2);
                val -= deprec;
                printf("Fin de año %2d", anual);
                printf("  Depreciación: %7.2f", deprec);
                printf("  Valor actual: %8.2f\n", val);
            }
            break;
        case 4:    /* fin de los cálculos */

            printf("\nHasta luego y que tenga un buen día\n");
            break;
        default:    /* generar mensaje de error */

            printf("\nEntrada de datos incorrecta");
            printf("-repite por favor\n");
        }   /* fin de switch */
    }   /* fin de while */
}
```

La forma de realizar los cálculos correspondientes al método de la suma de los dígitos de los años puede resultar algo oscura. En particular, el término `(n-anual+1)` en el numerador requiere alguna explicación. Esta cantidad se utiliza para contar *decrecientemente* (de `n` a `1`) mientras `anual` avanza `crecientemente` (de `1` a `n`). Estos valores declinantes son requeridos por el método de la suma de los dígitos de los años. Por supuesto, podríamos haber utilizado en lugar de esto un bucle con un contador que decreciese, esto es,

```
for (anual = n; anual >= 1; --anual)
```

pero entonces habríamos necesitado el correspondiente bucle con contador creciente que escribiese los resultados calculados anualmente. También el término `(n*(n+1)/2)` que aparece en el denominador es una fórmula para la suma de los `n` primeros dígitos, esto es, `1 + 2 + . . . + n`.

El programa está diseñado para ejecutarse de forma interactiva, con mensajes que solicitan los datos de entrada correspondientes. Observe que el programa genera un *menú* con cuatro opciones, para calcular la depreciación utilizando uno de los tres métodos o terminar la ejecución del programa. La computadora continuará aceptando nuevos conjuntos de datos de entrada y realizará los cálculos apropiados para cada uno de ellos, hasta que se seleccione la opción 4 del menú. El programa generará automáticamente un mensaje de error y volverá al menú si se introduce algún valor distinto de 1, 2, 3 o 4 en respuesta a la opción del menú.

Se muestra a continuación una sesión representativa del funcionamiento del programa. Se ha depreciado un objeto de 8000 dólares por un período de diez años utilizando cada uno de los tres métodos. También se presenta la respuesta con el mensaje de error a una introducción incorrecta de datos en la selección de opción en el menú. Finalmente, el cálculo termina como consecuencia de la última opción seleccionada en el menú.

```
Método: (1-LR  2-BDD  3-SDA  4-Fin) 1
Valor original: 8000
Número de años: 10

Método de línea recta

Fin de año  1  Depreciación:  800.00  Valor actual:  7200.00
Fin de año  2  Depreciación:  800.00  Valor actual:  6400.00
Fin de año  3  Depreciación:  800.00  Valor actual:  5600.00
Fin de año  4  Depreciación:  800.00  Valor actual:  4800.00
Fin de año  5  Depreciación:  800.00  Valor actual:  4000.00
Fin de año  6  Depreciación:  800.00  Valor actual:  3200.00
Fin de año  7  Depreciación:  800.00  Valor actual:  2400.00
Fin de año  8  Depreciación:  800.00  Valor actual:  1600.00
Fin de año  9  Depreciación:  800.00  Valor actual:   800.00
Fin de año 10  Depreciación:  800.00  Valor actual:     0.00

Método: (1-LR  2-BDD  3-SDA  4-Fin) 2
Valor original: 8000
Número de años: 10

Método de balance doblemente declinante

Fin de año  1  Depreciación: 1600.00  Valor actual:  6400.00
Fin de año  2  Depreciación: 1280.00  Valor actual:  5120.00
Fin de año  3  Depreciación: 1024.00  Valor actual:  4096.00
Fin de año  4  Depreciación:  819.20  Valor actual:  3276.80
Fin de año  5  Depreciación:  655.36  Valor actual:  2621.44
Fin de año  6  Depreciación:  524.29  Valor actual:  2097.15
Fin de año  7  Depreciación:  419.43  Valor actual:  1677.72
Fin de año  8  Depreciación:  335.54  Valor actual:  1342.18
Fin de año  9  Depreciación:  268.44  Valor actual:  1073.74
Fin de año 10  Depreciación:  214.75  Valor actual:  858.99
```

```
Método: (1-LR  2-BDD  3-SDA  4-Fin) 3
Valor original: 8000
Número de años: 10

Método de la suma de los dígitos de los años

Fin de año  1  Depreciación: 1454.55  Valor actual:  6545.45
Fin de año  2  Depreciación: 1309.09  Valor actual:  5236.36
Fin de año  3  Depreciación: 1163.64  Valor actual:  4072.73
Fin de año  4  Depreciación: 1018.18  Valor actual:  3054.55
Fin de año  5  Depreciación:  872.73  Valor actual:  2181.82
Fin de año  6  Depreciación:  727.27  Valor actual:  1454.55
Fin de año  7  Depreciación:  581.82  Valor actual:   872.73
Fin de año  8  Depreciación:  436.36  Valor actual:   436.36
Fin de año  9  Depreciación:  290.91  Valor actual:   145.45
Fin de año 10  Depreciación:  145.45  Valor actual:     0.00

Método: (1-LR  2-BDD  3-SDA  4-Fin) 5

Entrada de datos incorrecta - repite, por favor

Método: (1-LR  2-BDD  3-SDA  4-Fin) 4

Hasta luego y que tenga un buen día
```

Observe que el método de balance doblemente declinante y el de la suma de los dígitos de los años proporcionan una gran depreciación anual durante los primeros años, pero una mucho menor durante los últimos del tiempo de vida del objeto. Vemos también que el objeto tiene un valor cero al final de su tiempo de vida cuando se utiliza el método de línea recta y el de la suma de los dígitos de los años, pero queda un valor residual sin depreciar cuando se usa el método del balance doblemente declinante.

6.8. LA INSTRUCCIÓN `break`

La instrucción `break` se utiliza para terminar la ejecución de bucles o salir de una instrucción `switch`. Se puede utilizar dentro de una instrucción `while`, `do - while`, `for` o `switch`.

La instrucción `break` se puede escribir sencillamente de la siguiente forma:

```
break;
```

sin contener ninguna otra expresión o instrucción.

Ya hemos visto en algunos ejemplos el uso de la instrucción `break` dentro de una instrucción `switch`, en la sección 6.7. La instrucción `break` se ocupa de transferir el control fuera de la instrucción `switch` completa, a la primera instrucción que se encuentre a continuación de ella.

EJEMPLO 6.27. Consideremos de nuevo la instrucción `switch` presentada inicialmente en el Ejemplo 6.24.

```
switch (eleccion = toupper(getchar())) {

case 'R':
      printf("ROJO");
      break;

case 'B':
      printf("BLANCO");
      break;

case 'A':
      printf("AZUL");
      break;

default:
      printf("ERROR");
      break;
}
```

Observe que cada grupo de instrucciones finaliza con una instrucción `break`, con el fin de transferir el control fuera de la instrucción `switch`. La instrucción `break` es necesaria dentro de cada uno de los tres primeros grupos con el fin de evitar la ejecución de los siguientes grupos de instrucciones. El último grupo no requiere una instrucción `break`, ya que el control se transferirá fuera de la instrucción `switch` automáticamente después de la ejecución del último grupo de instrucciones. Sin embargo, la inclusión de la última instrucción `break` es una costumbre propia de un buen estilo de programación, con vistas a que esté ya presente en el caso de que se añadan otros grupos de instrucciones posteriormente.

Si se incluye una instrucción `break` en un bucle `while`, `do - while` o `for`, entonces se transfiere el control fuera del bucle en el momento en que se encuentre la instrucción `break`. Esto proporciona una forma conveniente de terminar un bucle cuando se detecta un error o alguna otra condición irregular.

EJEMPLO 6.28. He aquí algunas muestras de bucles que contienen instrucciones `break`. En cada situación, el bucle continuará su ejecución mientras el valor actual de la variable en coma flotante `x` no sea mayor que `100`. Sin embargo, se saldrá del bucle si se detecta un valor de `x` negativo.

Primero consideremos un bucle `while`.

```
scanf("%f", &x);
while (x <= 100) {
      if (x < 0) {
         printf("ERROR - VALOR NEGATIVO DE X");
         break;
      }

      /* procesar el valor no negativo de x */
      . . . . .
      scanf("%f", &x);
}
```

Consideremos ahora un bucle do - while que hace lo mismo.

```
do  {
      scanf("%f", &x);
      if (x < 0)  {
         printf("ERROR - VALOR NEGATIVO DE X");
         break;
    }

    /* procesar el valor no negativo de x */
    . . . . .
} while (x <= 100);
```

Finalmente, he aquí un bucle for semejante.

```
for (cont = 1; x <= 100; ++cont)  {
    scanf("%f", &x);
    if (x < 0)  {
       printf("ERROR - VALOR NEGATIVO DE X");
       break;
    }

    /* procesar el valor no negativo de x */
    . . . . .
}
```

En el caso de varias instrucciones while, do - while, for o switch anidadas, una instrucción break causará la transferencia de control fuera de la instrucción más interna en la que se encuentre, pero no fuera de las instrucciones externas. Ya hemos visto una muestra de esto en el Ejemplo 6.26, en donde se encuentra una instrucción switch dentro de una instrucción while. Se muestra otro ejemplo a continuación.

EJEMPLO 6.29. Consideremos el siguiente esquema de un bucle while incluido en uno for.

```
for (cont = 0; cont <= n; ++cont)  {
    . . . . .
    while (c = getchar() != '\n')  {
          if (c = '*') break;
          . . . . .
    }
}
```

Si la variable de carácter c tiene asignado un asterisco (*), entonces el bucle while terminará. Sin embargo, continuará la ejecución del bucle for. Por tanto, si el valor de cont es menor que n cuando se salga del bucle while, la computadora incrementará cont y hará otra pasada a través del bucle for.

6.9. LA INSTRUCCIÓN `Continue`

La instrucción `continue` se utiliza para *saltarse* el resto de la pasada actual a través de un bucle. El bucle *no* termina cuando se encuentra una instrucción `continue`. Sencillamente no se ejecutan las instrucciones que se encuentran a continuación en el mismo y se salta directamente a la siguiente pasada a través del bucle. (Observe esta diferencia importante entre `continue` y `break`.)

La instrucción `continue` se puede incluir dentro de una instrucción `while`, `do - while` o `for`. Simplemente se escribe así:

```
continue;
```

sin incluir otras instrucciones o expresiones.

EJEMPLO 6.30. He aquí algunas muestras de bucles que contienen instrucciones `continue`.

Primero consideraremos un bucle `do - while`.

```
do  {
        scanf("%f", &x);
        if (x < 0)  {
            printf("ERROR - VALOR NEGATIVO DE X");
            continue;

    };

    /* procesar el valor no negativo de x */

        . . . . .
} while (x <= 100);
```

He aquí un bucle `for` semejante.

```
for (cont = 1; x <= 100; ++cont)  {
     scanf("%f", &x);
     if (x < 0)  {
        printf("ERROR - VALOR NEGATIVO DE X");
        continue;
     }
     /* procesar el valor no negativo de x */
     . . . . .
}
```

En cada caso no se ejecutará la parte en la que se procesa el valor actual de `x` si éste es negativo. Se continuará en este caso con la siguiente pasada del bucle.

Es interesante comparar estas instrucciones de control con las mostradas en el Ejemplo 6.28, que hacen uso de la instrucción `break` en lugar de la `continue`. (¿Por qué no se incluye en este ejemplo una modificación del bucle `while` que aparece en el Ejemplo 6.28?)

EJEMPLO 6.31. Media de una lista de números no negativos. En el Ejemplo 6.17 vimos un programa en C que utilizaba un bucle `for` para calcular la media de una lista de `n` números. Modifiquemos este programa para que procese solamente los números no negativos.

El programa anterior requiere dos pequeños cambios para que se comporte de esta forma. Primero, el bucle `for` debe incluir una instrucción `if` que determine si cada valor de `x` es o no negativo. Segundo, necesitamos un contador especial (`nmedia`) para determinar cuántos números no negativos se han procesado. Este contador aparecerá en el denominador cuando se calcule la media (la media se calculará así: `media = suma/nmedia`).

Presentamos a continuación el programa en C. Es interesante compararlo con el que aparece en el Ejemplo 6.17.

```
/* calcular la media de los números no negativos de una lista de n
   números */

#include <stdio.h>

main()

{
   int n, cont , nmedia = 0;
   float x, media, suma = 0;

   /* inicializar y leer el valor de n */
   printf("¿Cuantos números? ");
   scanf("%d", &n);

   /* leer los números */
   for (cont = 1; cont <= n; ++cont)  {
        printf("x = ");
        scanf("%f", &x);
        if (x < 0) continue;
        suma += x;
        ++nmedia;
   }

   /* calcular la media y escribir la respuesta */
   media = suma/nmedia;
   printf("\nLa media es %f\n", media);
}
```

Cuando se ejecuta el programa con valores no negativos de `x`, se comporta exactamente igual que la anterior versión del Ejemplo 6.17. Sin embargo, cuando se asignan a `x` valores negativos, éstos son ignorados al calcular la media.

Mostramos a continuación una sesión interactiva a modo de ejemplo. Como de costumbre, las respuestas del usuario se encuentran subrayadas.

```
¿Cuantos números? 6
x = 1
x = -1
x = 2
x = -2
```

```
x = 3
x = -3

La media es 2.000000
```

Ésta es la media de los números positivos. Observe que si se hubiesen considerado todos los números la media habría sido cero.

6.10. EL OPERADOR COMA

Introducimos ahora el operador coma (,), que se utiliza principalmente en la instrucción `for`. Este operador permite que aparezcan dos expresiones en situaciones en donde sólo se utilizaría una expresión. Por ejemplo, es posible escribir

`for (expresión1a, expresión1b; expresión2; expresión3) instrucción`

en donde *expresión 1a* y *expresión 1b* son dos expresiones, separadas por el operador coma, en donde normalmente sólo aparecería una expresión (*expresión 1*). Estas expresiones se ocupan de inicializar dos índices, típicamente, que se utilizan simultáneamente dentro del bucle `for`.

Análogamente se puede utilizar en una instrucción `for` el operador coma de la siguiente forma:

`for (expresión1; expresión2; expresión3a, expresión3b) instrucción`

Aquí *expresión 3a* y *expresión 3b*, separadas por el operador coma, aparecen en lugar de la expresión única habitual. La forma frecuente de utilizar las dos expresiones es para alterar (incrementar o decrementar) los dos índices que se están utilizando simultáneamente dentro del bucle. Por ejemplo, un índice podría irse incrementando mientras el otro decrece.

EJEMPLO 6.32. Búsqueda de palíndromos. Un *palíndromo* es una palabra o una frase que se lee de la misma forma hacia delante que hacia atrás. Por ejemplo, las palabras *ala* y *rapar* son palíndromos. También lo es la frase *dábale arroz a la zorra el abad*, si no tenemos presentes espacios en blanco, signos de puntuación y el acento de «dábale».

Escribamos un programa en C que lea una línea de texto que contenga una palabra o una frase y determine si es o no un palíndromo. Para hacer esto, compararemos el primer carácter con el último, el segundo carácter con el penúltimo, y así sucesivamente, hasta que hayamos alcanzado el punto medio del texto. Las comparaciones en nuestro programa incluirán los signos de puntuación y los espacios en blanco.

Con el fin de esbozar el esquema del programa, definamos las siguientes variables:

`letras` = un array de caracteres de 80 elementos. Estos elementos serán los caracteres de la línea de texto.

`aux` = una variable entera que indicará el número de caracteres asignados a `letras`, sin incluir el carácter de escape \0 del final.

`cont` = una variable entera que se utilizará como índice al movernos hacia delante en `letras`.

`contr` = una variable entera que se utiliza como índice al movernos hacia atrás en `letras`.
`indicador` = una variable entera que se utilizará para indicar una condición de verdadero/falso. La condición verdadero indicará que se ha encontrado un palíndromo.
`bucle` = una variable entera cuyo valor es siempre igual a `1`, apareciendo, por tanto, siempre como verdadera. La intención de esto es continuar la ejecución del bucle principal hasta que una determinada condición indique su fin.

Podemos escribir ahora el siguiente esquema:

1. Definir las constantes simbólicas `EOL` («end of line» o fin de línea), `VERDADERO` y `FALSO`.
2. Declarar todas las variables e inicializar `bucle` (asignar `VERDADERO` a `bucle`).
3. Comenzar el bucle principal.
   a) Asignar `VERDADERO` a `indicador`, anticipando el hallazgo de un palíndromo.
   b) Leer la línea de texto carácter a carácter y almacenarla en `letras`.
   c) Comprobar si los tres primeros caracteres de la línea en mayúsculas son `F`, `I` y `N`, respectivamente. Si es así, salir del bucle principal y finalizar la ejecución del programa.
   d) Asignar a `aux` el valor final de `cont` menos `1`. Este valor indicará el número de caracteres que tiene la línea de texto, sin incluir el carácter de escape final `\0`.
   e) Comparar cada carácter de la primera mitad de `letras` con el correspondiente en la segunda mitad. Si se encuentra alguna falta de coincidencia, asignar `FALSO` a `indicador` y salir del bucle (más interno) de comparación.
   f) Si `indicador` es `VERDADERO`, escribir un mensaje que informe del hallazgo de un palíndromo. De otra forma, escribir un mensaje en el que se informe de que no se ha encontrado ningún palíndromo.
4. Repetir el paso 3 (hacer otra pasada por el bucle exterior), procesando otra línea de texto.

He aquí el pseudocódigo correspondiente.

```
#include archivos

#define constantes simbólicas

main()

{
   /* declarar e inicializar variables */

   while (bucle)  {

      indicador = VERDADERO;     /* anticipar palíndromo */

      /* leer una línea de texto y almacenarla en letras */

      /* salir del bucle while si los tres primeros caracteres
         de letras son FIN (comprobarlos en mayúsculas) */

      /* asignar el número de caracteres del texto a aux */

      for ((cont = 0, contr = aux); cont <= (aux - 1)/2;
             (++cont, --contr))  {
```

```
            if (letras[cont] != letras[contr])   {
               indicador = FALSO;

               /* no es un palíndromo - salir del bucle for */
            }
        }

        /* escribir un mensaje indicando si letras contiene o
           no un palíndromo */
    }
}
```

El programa utiliza el operador coma en una instrucción `for` para comparar cada carácter de la primera mitad de `letras` con el carácter correspondiente de la segunda mitad. Por tanto, mientras `cont` va tomando valores de 0 a `(aux - 1) / 2`, `contr` lo hace de `aux` a `(aux / 2) + 1`. Observe que se realiza la división entera (obteniéndose el cociente truncado) para establecer estos valores límite.

Observe también que hay dos operadores coma dentro de la instrucción `for`. Cada operador coma y sus operandos asociados se encuentran entre paréntesis. Esto no es necesario, pero recalca el hecho de que cada par de operandos forman un argumento dentro de la instrucción `for`.

A continuación se muestra el programa en C completo.

```
/* buscar un palíndromo */

#include <stdio.h>
#include <ctype.h>

#define EOL     '\n'
#define VERDADERO 1
#define FALSO   0

main()
{
  char letras[80];
  int aux, cont, contr, indicador, bucle = VERDADERO;

  /* bucle principal */

  while (bucle)    {
     indicador = VERDADERO;

     /* leer el texto */

     printf("Introduce una palabra o frase debajo:\n");
     for (cont = 0; (letras[cont] = getchar()) != EOL; ++cont)
        ;
     if ((toupper(letras[0])  ==  'F') &&
        (toupper(letras[1])  ==  'I') &&
        (toupper(letras[2])  ==  'N')) break;
     aux = cont - 1;

     /* realizar la búsqueda */
```

```
        for ((cont = 0, contr = aux); cont <= aux/2;
             (++cont, --contr))   {

                 if (letras[cont] != letras[contr])   {
                    indicador = FALSO;
                    break;
                 }
        }

        /* escribir mensaje */

        for (cont = 0; cont <= aux; ++cont)
               putchar(letras[cont]);
        if (indicador) printf(" ES un palíndromo\n\n");
        else printf(" NO ES un palíndromo\n\n");
    }
}
```

A continuación se muestra una sesión interactiva típica, detallándose la salida generada cuando se ejecuta el programa. Como se ha hecho hasta el momento, las respuestas del usuario se encuentran subrayadas.

```
Introduce una palabra o frase debajo:
RAPAZ

RAPAZ NO ES un palíndromo

Introduce una palabra o frase debajo:
RAPAR

RAPAR ES un palíndromo

Introduce una palabra o frase debajo:
salas

salas ES un palíndromo

Introduce una palabra o frase debajo:
DABALE ARROZ A LA ZORRA EL ABAD

DABALE ARROZ A LA ZORRA EL ABAD NO ES un palíndromo

Introduce una palabra o frase debajo:
DABA LE ARROZ ALA ZORRA EL ABAD

DABA LE ARROZ ALA ZORRA EL ABAD ES un palíndromo

Introduce una palabra o frase debajo:
FIN
```

Recuerde que el operador coma acepta dos expresiones como operandos. Estas expresiones se evaluarán de izquierda a derecha. En situaciones que requieren la evaluación de la expresión

en conjunto (la expresión formada por los dos operandos y el operador coma), el tipo y el valor de la expresión en conjunto serán determinadas por el operando de la derecha.

Dentro del conjunto de operadores de C, el operador coma tiene la precedencia menor de todos. Por tanto, el operador coma se encuentra solo dentro de su grupo de precedencia, por debajo del grupo de precedencia integrado por los distintos operadores de asignación (ver Apéndice C). Su asociatividad es de izquierda a derecha.

6.11. LA INSTRUCCIÓN `goto`

La instrucción `goto` se utiliza para alterar la secuencia de ejecución normal del programa, transfiriéndose el control a otra parte de él. En su forma general, la instrucción `goto` se escribe

```
goto etiqueta;
```

en donde *`etiqueta`* es un identificador que se utiliza para rotular la instrucción a la que se transferirá el control.

Se puede transferir el control a cualquier otra instrucción del programa. (Para ser más precisos, se puede transferir el control a cualquier lugar dentro de la *función* actual. Introduciremos y trataremos las funciones en el capítulo siguiente.) La instrucción por la que se continuará la ejecución debe encontrarse etiquetada, y la etiqueta debe ir seguida de dos puntos (:). Por tanto, la instrucción etiquetada aparecerá de la siguiente forma:

```
etiqueta: instrucción
```

Cada instrucción etiquetada dentro de un programa (más concretamente, dentro de la función actual) debe tener una única etiqueta; es decir, dos instrucciones no pueden tener la misma etiqueta.

EJEMPLO 6.33. El siguiente esquema de programa muestra cómo se puede utilizar la instrucción `goto` para transferir el control fuera de un bucle si se da una condición inesperada.

```
/* bucle principal */

scanf("%f", &x);
while (x <= 100)  {
      . . . . .
      if (x < 0) goto error;
      . . . . .
      scanf ("%f", &x);
}
      . . . . .

/* rutina de detección de error */

error:  {
          printf("ERROR - VALOR NEGATIVO DE X");
              . . . . .
        }
```

En este ejemplo se transfiere el control fuera del bucle `while`, a la instrucción compuesta con etiqueta `error`, si se detecta un valor negativo de la variable `x`.

Se podría haber hecho lo mismo utilizando la instrucción `break`, como se vio en el Ejemplo 6.28. De hecho es preferible el uso de la instrucción `break`. La utilización de la instrucción `goto` se presenta aquí a título meramente ilustrativo.

Todos los lenguajes de propósito general populares poseen la instrucción `goto`, aunque actualmente todas las técnicas de programación instan a evitar su utilización. En algunos de los lenguajes más antiguos, como Fortran o BASIC, la instrucción `goto` se utiliza con gran profusión. Las aplicaciones más comunes son las siguientes:

1. Ejecutar condicionalmente instrucciones o grupos de instrucciones bajo determinadas circunstancias.
2. Saltar al fin de un bucle en determinadas condiciones, no ejecutándose, por tanto, el resto del bucle en la pasada actual.
3. Finalizar totalmente la ejecución de un bucle bajo determinadas condiciones.

Los elementos estructurados del C permiten realizar todas estas operaciones sin recurrir a la instrucción `goto`. Por ejemplo, la ejecución condicional se puede realizar mediante la instrucción `if - else`; saltar al final de un bucle se puede realizar mediante la instrucción `continue`; y terminar la ejecución de un bucle mediante la instrucción `break`. Es preferible la utilización de esos elementos estructurados al de la instrucción `goto` porque el uso de `goto` tiende a favorecer (o al menos, a no disminuir) que se disperse la lógica subyacente por todo el programa, mientras que los elementos estructurados de C requieren que todo el programa se escriba en una forma ordenada y secuencial. Por esta razón, *la utilización de la instrucción* `goto` *se debe evitar, como norma general, dentro de los programas en C.*

Sin embargo, a veces se presentan situaciones de forma esporádica en las que la instrucción `goto` puede ser muy útil. Consideremos, por ejemplo, una situación en la que se necesita salir de un bucle doblemente anidado al detectar determinada condición. Esto se puede hacer mediante dos instrucciones `if - break`, una dentro de cada bucle, aunque esto es un tanto enrevesado. Una solución mejor en este caso particular podría ser el utilizar una instrucción `goto` para transferir el control fuera de ambos bucles de una vez. Este procedimiento se ilustra en el ejemplo siguiente.

EJEMPLO 6.34. Conversión de varias líneas de texto a mayúsculas. El Ejemplo 6.19 presenta un programa que convierte varias líneas de texto a mayúsculas, procesando una línea de texto cada vez, hasta que el primer carácter de una línea sea un asterisco (`*`). Modifiquemos ahora el programa para detectar una con-dición de parada, indicada por la presencia de dos signos de dólar consecutivos (`$$`) en cualquier parte den-tro de una línea de texto. Si se encuentra la condición de parada, el programa imprimirá la línea de texto con los signos de dólar incluidos, seguida de un mensaje adecuado. En ese momento terminará la ejecución del programa.

La lógica del programa será la misma que la dada en el Ejemplo 6.19, pero añadiremos un bucle para comprobar si se encuentran los dos signos de dólar consecutivos. Por tanto, el programa procederá como sigue.

1. Asignar un valor inicial de `1` al índice del bucle más externo (`contlineas`).
2. Efectuar los siguientes pasos de forma repetida, para las sucesivas líneas de texto, mientras el primer carácter de la línea no sea un asterisco.
   a) Leer una línea de texto y asignar los caracteres a los elementos del array de caracteres `letras`. Una línea se definirá como una sucesión de caracteres terminada por una indicación de final de la línea (un carácter de *nueva línea*).

   b) Asignar la cuenta de caracteres, incluido el de nueva línea, a `aux`.
   c) Escribir la línea en mayúsculas, utilizando la función de biblioteca `toupper` para realizar la conversión. Escribir a continuación dos caracteres de nueva línea (para que la siguiente línea de entrada se encuentre separada de la salida actual por una línea en blanco) e incrementar el contador de líneas (`contlineas`).
   d) Buscar en todos los caracteres de la línea la presencia de dos signos de dólar consecutivos. Si se detectan, escribir un mensaje informando de la detección de la condición de parada y saltar al final del programa.

3. Una vez que se ha detectado un asterisco al comienzo de una línea, escribir «Hasta luego», y terminar la ejecución.

He aquí el programa completo en C.

```
/* convertir varias líneas de texto a mayúsculas

   Continuar la conversión hasta que el primer carácter de una
   línea sea un asterisco (*). Finalizar el programa si se
   encuentran dentro de una línea dos signos de dólar ($$).   */

#include <stdio.h>
#include <ctype.h>

#define EOL '\n'

main()

{
   char letras[80];
   int aux, cont, contlineas = 1;

   while ((letras[0] = getchar()) != '*')  {

          /* leer la línea de texto */
          for (cont = 1; (letras[cont] = getchar()) != EOL; ++cont)
               ;
          aux = cont;

          /* escribir la línea de texto */
          for (cont = 0; cont < aux; ++cont)
               putchar(toupper(letras[cont]));
          printf("\n\n");
          ++contlineas;

          /* comprobar la condición de salida */
          for (cont = 1; cont < aux; ++cont)
              if (letras[cont-1] == '$' && letras[cont] == '$') {
                 printf("CONDICION DE SALIDA DETECTADA - ");
                 printf("FIN DE EJECUCION\n\n");
                 goto fin;
              }
   }
   fin: printf("Hasta luego");
}
```

Es interesante comparar este programa con el correspondiente presentado anteriormente en el Ejemplo 6.19. El programa actual contiene un bucle `for` adicional dentro del bucle `while`, al final. Este bucle `for` examina pares consecutivos de caracteres, buscando la condición de parada (`$$`), después de haber escrito la línea completa en mayúsculas. Si se encuentra una condición de parada, entonces se transfiere el control a la instrucción `printf` final (`«Hasta luego»`) que se ha etiquetado ahora con `fin`. Observe que esta transferencia de control causa una salida del bucle `for` actual y del bucle exterior `while` desde la instrucción `if`.

El lector puede ejecutar este programa y utilizar las dos condiciones, la de terminación normal (un asterisco al comienzo de una línea) y la de parada. Compárense los resultados obtenidos con la salida mostrada en el Ejemplo 6.19.

CUESTIONES DE REPASO

6.1. ¿Qué es ejecución condicional?

6.2. ¿Qué es selección?

6.3. ¿Qué es un bucle? Describir dos formas diferentes de construir un bucle.

6.4. Citar todas las reglas relacionadas con la utilización de los cuatro operadores relacionales, los dos operadores de igualdad, las dos conectivas lógicas y el operador unario de negación. ¿Qué tipos de operandos se utilizan con cada operador?

6.5. ¿Cómo se interpretan las variables y constantes de tipo carácter cuando se utilizan como operandos de un operador relacional?

6.6. ¿En qué difieren las instrucciones de expresión de las instrucciones compuestas? Mencionar las reglas asociadas a cada una.

6.7. ¿Cuál es el propósito de la instrucción `if - else`?

6.8. Describir las dos formas diferentes de la instrucción `if - else`. ¿En qué se diferencian?

6.9. Comparar el uso de la instrucción `if - else` con el del operador `?:`. ¿De qué forma se puede utilizar el operador `?:` en lugar de una instrucción `if - else`?

6.10. Citar las reglas sintácticas asociadas con la instrucción `if - else`.

6.11. ¿Cómo se interpretan las instrucciones `if - else` anidadas? En particular, ¿cómo se interpreta la siguiente?

```
if e1 if e2 s1
      else s2
```

¿Qué expresión lógica se encuentra asociada a la cláusula `else`?

6.12. ¿Qué ocurre cuando se encuentra una expresión con valor no nulo dentro de un grupo de instrucciones `if - else` anidadas?

6.13. ¿Cuál es la finalidad de la instrucción `while`? ¿Cuándo es evaluada la expresión lógica? ¿Cuál es el mínimo número de veces que se puede ejecutar un bucle `while`?

6.14. ¿Cómo finaliza la ejecución de un bucle `while`?

6.15. Hacer una relación de las reglas sintácticas asociadas a la instrucción `while`.

6.16. ¿Cuál es la finalidad de la instrucción `do - while`? ¿En qué difiere de la instrucción `while`?

6.17. ¿Cuál es el mínimo número de veces que se puede ejecutar un bucle `do - while`? Compararlo con un bucle `while` y explicar las razones por las que se diferencian.

6.18. Mencionar las reglas sintácticas asociadas con la instrucción `do - while`. Compararlas con las de la instrucción `while`.

6.19. ¿Cuál es la finalidad de la instrucción `for`? ¿En qué se distingue de las instrucciones `while` y `do - while`?

6.20. ¿Cuántas veces se ejecutará un bucle `for`? Compararlo con los bucles `while` y `do - while`.

6.21. ¿Cuál es la finalidad del índice de una instrucción `for`?

6.22. ¿Se puede omitir alguna de las tres expresiones iniciales en una instrucción `for`? Si es así, ¿cuáles son las consecuencias de cada omisión?

6.23. Citar las reglas sintácticas asociadas con la instrucción `for`.

6.24. ¿Qué reglas se aplican a los bucles anidados? ¿Se puede anidar un tipo de bucle en otro de tipo distinto?

6.25. ¿Se pueden incluir bucles en las instrucciones `if - else`? ¿E instrucciones `if - else` en los bucles?

6.26. ¿Cuál es la finalidad de la instrucción `switch`? ¿En qué difiere esta instrucción de las otras descritas en este capítulo?

6.27. ¿Qué son las etiquetas «case» (o prefijos «case»)? ¿Qué tipo de expresiones se deben utilizar para representar una etiqueta «case»?

6.28. Mencionar las reglas sintácticas asociadas al uso de la instrucción `switch`. ¿Pueden estar asociadas varias etiquetas «case» a un único grupo de instrucciones?

6.29. ¿Qué ocurre cuando el valor de la expresión en la instrucción `switch` coincide con el de una de las etiquetas? ¿Qué ocurre cuando el valor de esta expresión no coincide con el de ninguna de las etiquetas?

6.30. ¿Se puede definir una opción por omisión dentro de una instrucción `switch`? Si es así, ¿cómo se debe etiquetar la opción por omisión?

6.31. Comparar la utilización de la instrucción `switch` con la de instrucciones `if - else` anidadas. ¿Cuál es más conveniente?

6.32. ¿Cuál es la finalidad de la instrucción `break`? ¿Dentro de qué instrucciones de control se puede incluir la instrucción `break`?

6.33. Supóngase que una instrucción `break` se encuentra incluida en la más interna de una serie de instrucciones de control anidadas. ¿Qué ocurre cuando se ejecuta la instrucción `break`?

6.34. ¿Cuál es el propósito de la instrucción `continue`? ¿Dentro de qué instrucciones de control se puede incluir la instrucción `continue`? Compararla con la instrucción `break`.

6.35. ¿Cuál es la finalidad del operador coma? ¿Dentro de qué instrucciones de control suele aparecer el operador coma?

6.36. En situaciones que requieren la evaluación de una expresión que contiene el operador coma, ¿qué operando determinará el tipo y el valor de toda la expresión (la expresión a la izquierda o a la derecha de la coma)?

6.37. ¿Cuál es la precedencia del operador coma respecto a los otros operadores de C?

6.38. ¿Cuál es la finalidad de la instrucción `goto`? ¿Cómo se identifica la instrucción etiquetada asociada a ella?

6.39. ¿Existe alguna restricción respecto al lugar al que se puede transferir el control dentro de un programa en C?

6.40. Relacionar las reglas sintácticas asociadas a la instrucción `goto`.

6.41. Comparar la sintaxis relacionada con las etiquetas de las instrucciones (correspondientes a `goto`) y la de las etiquetas «case» (prefijos «case»).

6.42. ¿Por qué se debe evitar el uso de la instrucción `goto`? ¿En qué ocasiones puede ser útil esta instrucción? ¿Qué tipo de usos se deben evitar y por qué? Discutirlo con detalle.

PROBLEMAS

6.43. Explicar qué ocurre cuando se ejecuta la siguiente instrucción.

```
if (abs(x) < xmin) x = (x > 0) ? xmin : -xmin;
```

¿Es ésta una instrucción compuesta? ¿Se encuentra incluida en esta instrucción una instrucción compuesta?

6.44. Identificar todas las instrucciones compuestas que aparecen en el siguiente fragmento de programa.

```
{
  suma = 0;
  do  {
         scanf("%d", &i);
         if (i < 0)  {
             i = -i;
             ++indicador;
         }
         suma += i;
  } while (i != 0);
}
```

6.45. Escribir un bucle que calcule la suma de cada tercer entero, comenzando por `i=2` (es decir, calcular la suma de `2 + 5 + 8 + 11 + ...`) para todos los valores de `i` menores que `100`. Escribir el bucle de tres formas diferentes:

a) Utilizando una instrucción `while`.
b) Utilizando una instrucción `do - while`.
c) Utilizando una instrucción `for`.

6.46. Repetir el Problema 6.45 calculando la suma de cada entero n-ésimo, comenzando por el valor asignado a `ncom` (es decir, `i = ncom`, `ncom + n`, `ncom + 2*n`, `ncom + 3*n`, y así sucesivamente). Continuar el proceso para todos los valores de `i` que no sean mayores que `nfin`.

6.47. Escribir un bucle que examine cada carácter en un array de caracteres llamado `texto` y escriba el equivalente ASCII (el valor numérico) de cada carácter. Supóngase que se especifica por adelantado el número de caracteres del array en la variable entera `n`. Escribir el bucle de tres formas diferentes:

a) Utilizando una instrucción `while`.
b) Utilizando una instrucción `do - while`.
c) Utilizando una instrucción `for`.

6.48. Repetir el Problema 6.47 suponiendo que no se especifica por adelantado el número de caracteres del array. La ejecución del bucle debe repetirse hasta que se encuentre un asterisco (`*`). Escribir el bucle de tres formas diferentes, como antes.

6.49. Generalizar el Problema 6.45 para generar una *serie* de bucles, donde cada bucle ha de calcular la suma de los enteros j-ésimos, en donde `j` se encuentra entre `2` y `13`. Comenzar cada bucle con `i=2` e incrementar `i` en `j` unidades hasta que `i` alcance el mayor valor posible menor que `100`. (En otras palabras, el primer bucle calculará la suma `2 + 4 + 6 + ... + 98`; el segundo bucle la suma `2 + 5 + 8 + ... + 98`; el tercero la suma `2 + 6 + 10 + ... + 98`, y así sucesivamente. El último bucle calculará la suma `2 + 15 + 28 + ... + 93`.) El programa debe escribir el valor de cada suma completa.

Utilizar bucles anidados para resolver este problema. Calcular cada suma en el bucle interior y hacer que el bucle exterior se ocupe de controlar el valor de `j` utilizado en cada pasada por el bucle interior. Utilizar una instrucción `for` para el bucle exterior, y para el bucle interior utilizar cada una de las tres instrucciones de repetición `while`, `do - while` y `for`. Redactar una solución por separado para cada tipo de bucle interior.

6.50. Escribir un bucle que genere enteros de tres en tres, comenzando por `i=2` hasta el valor máximo menor que `100`. Calcular la suma de los enteros generados que sean divisibles por `5`. Utilizar dos métodos distintos para comprobar esto último:

a) Utilizar el operador condicional `(?:)`.
b) Utilizar una instrucción `if - else`.

6.51. Generalizar el Problema 6.50 para que genere los enteros n-ésimos, comenzando por `ncom` (esto es, `i = ncom, ncom + n, ncom + 2*n, ncom + 3*n`, etc.). Continuar el proceso para todos los valores de `i` menores que `nfin`. Calcular la suma de los enteros que sean divisibles por `k`, en donde `k` representa cualquier entero positivo.

6.52. Escribir un bucle que examine cada carácter de un array de caracteres llamado `texto` y determine cuántos de los caracteres son letras, cuántos son dígitos, cuántos caracteres de espaciado y cuántos son otros tipos de caracteres (por ejemplo caracteres de puntuación). Suponga que `texto` contiene `80` caracteres.

6.53. Escribir un bucle que examine cada carácter de un array de caracteres llamado `texto` y determine cuántos de los caracteres son vocales y cuántos son consonantes. (*Sugerencia:* Determinar primero cuando un carácter es letra y, si es así, determinar el tipo de letra.) Suponga que `texto` contiene `80` caracteres.

6.54. Escribir una instrucción `switch` que examine el valor de una variable entera llamada `indicador` y escriba uno de los siguientes mensajes dependiendo de su valor:

a) `CALOR`, si `indicador` tiene el valor `1`
b) `TEMPLADO`, si `indicador` tiene el valor `2`
c) `FRIO`, si `indicador` tiene el valor `3`
d) `FUERA DE RANGO`, si `indicador` tiene el valor `4`

6.55. Escribir una instrucción `switch` que examine el valor de una variable de tipo carácter llamada `color` y escriba uno de los siguientes mensajes dependiendo de su valor:

a) `ROJO`, si `color` tiene asignado `r` o `R`,
b) `VERDE`, si `color` tiene asignado `v` o `V`,
c) `AZUL`, si `color` tiene asignado `a` o `A`,
d) `NEGRO`, si `color` tiene asignado cualquier otro carácter.

6.56. Escribir una estructura de control que examine el valor de una variable en coma flotante llamada `temp` y escriba uno de los siguientes mensajes dependiendo de su valor:

a) `HIELO`, si el valor de `temp` es menor que `0`.
b) `AGUA`, si el valor de `temp` se encuentra entre `0` y `100`.
c) `VAPOR`, si el valor de `temp` es mayor que `100`.

¿Se puede utilizar una instrucción `switch` en este problema?

6.57. Escribir un bucle `for` que lea los caracteres de un array de caracteres llamado `texto` y los escriba en sentido opuesto en otro array llamado `inverso`. Suponga que la formación `texto` contiene `80` caracteres. Utilizar el operador coma dentro de la instrucción `for`.

6.58. Describir la salida que generará cada uno de los siguientes programas en C.

a)
```
#include <stdio.h>

main()
{
     int i = 0, x = 0;

     while (i < 20)   {
        if (i % 5 == 0)   {
           x += i;
           printf("%d ", x);
        }
        ++i;

     }
     printf("\nx = %d", x);
}
```

b)
```
#include <stdio.h>

main()

{
     int i = 0, x = 0;

     do   {
             if (i % 5 == 0)   {
                x++;
                printf("%d ", x);
             }
             ++i;
     }  while (i < 20);
     printf("\nx = %d", x);
}
```

c)
```
#include <stdio.h>

main()

{
     int i = 0, x = 0;

     for (i = 1; i < 10; i *= 2)  {
          x++;
          printf("%d ", x);
     }
     printf("\nx = %d", x);
}
```

d)
```
#include <stdio.h>

main()

{
     int i = 0, x = 0;

     for (i = 1; i < 10; ++i)  {
          if (i % 2 == 1)
             x += i;
          else
             x--;
          printf("%d ", x);
     }
     printf("\nx = %d", x);
}
```

e)
```
#include <stdio.h>

main()

{
     int i = 0, x = 0;

     for (i = 1; i < 10; ++i)  {
          if (i % 2 == 1)
             x += i;
          else
             x--;
          printf("%d ", x);
          continue;
     }
     printf("\nx = %d", x);
}
```

f)
```
#include <stdio.h>

main()

{
     int i = 0, x = 0;

     for (i = 1; i < 10; ++i)  {
         if (i % 2 == 1)
            x += i;
         else
            x--;
         printf("%d ", x);
         break;
   }
     printf("\nx = %d", x);

}
```

g)
```
#include <stdio.h>

main()

{
     int i, j, x = 0;

     for (i = 0; i < 5; ++i)
         for (j = 0; j < i; ++j)  {
            x += (i + j - 1);
            printf("%d ", x);
        }
     printf("\nx = %d", x);
}
```

h)
```
#include <stdio.h>

main()

{
     int i, j, x = 0;

     for (i = 0; i < 5; ++i)
         for (j = 0; j < i; ++j)  {
            x += (i + j - 1);
            printf("%d ", x);
            break;
         }
     printf("\nx = %d", x);
}
```

i)
```
#include <stdio.h>

main()

{
     int i, j, x = 0;

     for (i = 0; i < 5; ++i)  {
          for (j = 0; j < i; ++j)
               x += (i + j - 1);
               printf("%d ", x);
               break;
          }
     printf("\nx = %d", x);
}
```

j)
```
#include <stdio.h>

main()

{
     int i, j, k, x = 0;

     for (i = 0; i < 5; ++i)
          for (j = 0; j < i; ++j)  {
               k = (i + j - 1);
               if (k % 2 == 0)
                  x += k;
               else
                  if (k % 3 == 0)
                     x += k - 2;
               printf("%d ", x);
          }
     printf("\nx = %d", x);
}
```

k)
```
#include <stdio.h>

main()

{
     int i, j, k, x = 0;

     for (i = 0; i < 5; ++i)
          for (j = 0; j < i; ++j)  {

               switch (i + j - 1)  {
```

```
                case -1:
                case  0:
                      x += 1;
                      break;

                case  1:
                case  2:
                case  3:
                      x += 2;
                      break;

                default:
                      x += 3;
                }
                printf("%d ", x);
            }
        printf("\nx = %d", x);
    }
```

l)
```
#include <stdio.h>

main()

{

    int i, j, k, x = 0;

    for (i = 0; i < 5; ++i)
        for (j = 0; j < i; ++j)  {

            switch (i + j - 1)  {

            case -1:
            case  0:
                  x += 1;
                  break;

            case  1:
            case  2:
            case  3:
                  x += 2;

            default:
                  x += 3;
            }
            printf("%d ", x);
        }
    printf("\nx = %d", x);
}
```

PROBLEMAS DE PROGRAMACIÓN

6.59. Modificar los programas dados en los Ejemplos 6.9, 6.12 y 6.16 para que cada programa haga lo siguiente:

a) Leer una línea de texto en mayúsculas, almacenarla en un array adecuado y escribirla a continuación en minúsculas.

b) Leer una línea de texto en mayúsculas y minúsculas, almacenarla en un array adecuado y a continuación escribirla con las mayúsculas y las minúsculas intercambiadas, todos los dígitos reemplazados por ceros y el resto de los caracteres sustituidos por asteriscos (*).

6.60. Compilar y ejecutar los programas dados en los Ejemplos 6.10, 6.13 y 6.17 utilizando los diez números siguientes:

27.5, 13.4, 53.8, 29.2, 74.5, 87.0, 39.9, 47.7, 8.1, 63.2

6.61. Compilar y ejecutar el programa dado en el Ejemplo 6.31 utilizando los diez números siguientes:

27.5, –13.4, 53.8, –29.2, 74.5, 87.0, 39.9, –47.7, –8.1, 63.2

Comparar los resultados calculados con los obtenidos en el último problema.

6.62. Modificar el programa dado en el Ejemplo 6.10 de forma que no se especifique por adelantado el tamaño de la lista de números. Continuar la ejecución del bucle (leyendo un nuevo valor de `x` y añadiéndolo a `suma`) hasta que se introduzca el valor cero. Por tanto, `x = 0` indicará la condición de fin.

6.63. Repetir el Problema 6.62 para el programa del Ejemplo 6.17.

6.64. Reescribir el programa de depreciación dado en el Ejemplo 6.26 utilizando la instrucción `if - else` en lugar de la instrucción `switch`. Comprobar el programa con los datos dados en el Ejemplo 6.26. ¿Qué versión se prefiere? ¿Por qué?

6.65. La ecuación

$$x^5 + 3x^2 - 10 = 0$$

que se presentó en el Ejemplo 6.22, se puede reescribir de la forma

$$x = [(10 - x^5)/3]^{1/2}$$

Reescribir el programa del Ejemplo 6.22 utilizando la ecuación en esta última forma. Ejecutar el programa y comparar los resultados calculados con los que aparecen en el Ejemplo 6.22. ¿Por qué son diferentes los resultados? (¿Generan siempre las computadoras respuestas correctas?)

6.66. Modificar el programa del Ejemplo 6.22, que calcula las raíces de una ecuación algebraica, reemplazando la instrucción `while` por una `do - while`. ¿Qué estructura se adapta mejor a este tipo de problema?

6.67. Modificar el programa del Ejemplo 6.22, que calcula las raíces de una ecuación algebraica, reemplazando la instrucción `while` por una `for`. Comparar el uso de las instrucciones `for`, `while` y `do - while`. ¿Qué versión se prefiere? ¿Por qué?

6.68. Añadir una rutina de detección de errores semejante a la del Ejemplo 6.21 al programa de cálculo de la depreciación del Ejemplo 6.26. La rutina debe generar mensajes de error seguidos de una nueva petición de entrada de los datos cuando se detecte un valor no positivo.

6.69. Escribir un programa completo en C para cada uno de los problemas que se presentan a continuación. Utilizar el tipo de instrucción de control más natural en cada caso. Comenzar con un esquema detallado; reescribirlo en forma de pseudocódigo si el traducirlo directamente a C no es inmediato. Asegurarse de utilizar un buen estilo de programación (comentarios, sangrados, etc.).

a) Calcular la *media ponderada* de una lista de *n* números, utilizando la fórmula

$$x_{media} = f_1x_1 + f_2x_2 + \dots + f_nx_n$$

donde las f son los pesos de cada cantidad y han de ser fraccionarios, es decir:

$$0 \leq f_i < 1 \quad \text{y} \quad f_1 + f_2 + \dots + f_n = 1$$

Comprobar el programa con los siguientes datos:

$i = 1$	$f = 0.06$	$x = 27.5$
2	0.08	13.4
3	0.08	53.8
4	0.10	29.2
5	0.10	74.5
6	0.10	87.0
7	0.12	39.9
8	0.12	47.7
9	0.12	8.1
10	0.12	63.2

b) Calcular el producto de una lista de *n* números. Comprobar el programa utilizando el siguiente conjunto de seis datos: 6,2, 12,3, 5,0, 18,8, 7,1, 12,8.

c) Calcular la *media geométrica* de una lista de números, utilizando la fórmula

$$x_{media} = [x_1x_2x_3 \dots x_n]^{1/n}$$

Comprobar el programa utilizando los valores de x dados en la parte *b*). Comparar los resultados obtenidos con la media aritmética de los mismos datos. ¿Qué media es mayor?

d) Determinar las raíces de la ecuación de segundo grado

$$ax^2 + bx + c = 0$$

utilizando la fórmula

$$x = \frac{-b \pm (b^2 - 4ac)^{1/2}}{2a}$$

(Ver Ejemplo 5.6). Permitir la posibilidad de que alguna de las constantes tenga el valor cero y que la cantidad $b^2 - 4ac$ sea menor o igual a cero. Comprobar el programa utilizando los siguientes conjuntos de datos:

$a = 2$	$b = 6$	$c = 1$
3	3	0
1	3	1
0	12	−3
3	6	3
2	−4	3

e) Los *números de Fibonacci* son los miembros de una interesante secuencia en la que cada número es igual a la suma de los dos números anteriores. En otras palabras,

$$F_i = F_{i-1} + F_{i-2}$$

en donde F_i es el i-ésimo número de Fibonacci. Los dos primeros números de Fibonacci son, por definición, iguales a 1, es decir:

$$F_1 = F_2 = 1.$$

Por tanto,

$$F_3 = F_2 + F_1 = 1 + 1 = 2$$
$$F_4 = F_3 + F_2 = 2 + 1 = 3$$
$$F_5 = F_4 + F_3 = 3 + 2 = 5$$

y así sucesivamente.

Escribir un programa en C que determine los n primeros números de Fibonacci. Comprobar el programa con $n = 7$, $n = 10$, $n = 17$ y $n = 23$.

f) Un *número primo* es una cantidad entera que es divisible (sin resto) sólo por 1 y por sí mismo. Por ejemplo, 7 es un número primo, pero 6 no lo es.

Calcular y presentar una lista con los n primeros números primos. (*Sugerencia:* un número n será número primo si los restos de las ecuaciones $n/2$, $n/3$, $n/4$, ..., $n^{1/2}$ son todos no nulos.) Comprobar el programa haciendo que calcule los 100 primeros números primos.

g) Escribir un programa interactivo que lea un valor entero positivo y determine lo siguiente:

i) Si el entero es un número primo.
ii) Si el entero es un número de Fibonacci.

Escribir el programa de forma que se ejecute repetidamente, hasta que se introduzca como valor de entrada un cero. Comprobar el programa con varios enteros.

h) Calcular la suma de los primeros n números impares ($1 + 3 + 5 + ... + 2n - 1$). Comprobar el programa con la suma de los 100 primeros números impares (nótese que el último entero será 199).

i) Se puede calcular el seno de x de forma aproximada sumando los n primeros términos de la serie infinita

$$\text{sen } x = x - x^3/3! + x^5/5! - x^7/7! + ...$$

en donde x se expresa en radianes. (*Nota*: π radianes = 180°).

Escribir un programa en C que lea el valor de x y calcule su seno. Escribir el programa de dos formas diferentes:

i) Sumar los n primeros términos, en donde n es un entero positivo que se introduce además del valor de x.
ii) Continuar sumando términos de la serie hasta que el valor del término siguiente sea menor (en valor absoluto) que 10^{-5}.

Comprobar el programa para $x = 1$, $x = 2$ y $x = -3$. Escribir en cada caso el número de términos utilizados para conseguir la respuesta final.

j) Supongamos que se consigue un préstamo de P dólares de un banco, con el reconocimiento de que se devolverán A dólares cada mes hasta que se haya completado la cantidad total prestada. Parte del pago mensual serán intereses, calculados como el i por ciento de la cantidad aún no pagada. El resto del pago servirá para reducir la cantidad adeudada.

Escribir un programa en C que determine la siguiente información:

i) La cantidad de interés pagado por mes.
ii) La cantidad de dinero aplicado a la reducción de la deuda total cada mes.
iii) La cantidad total de intereses que se lleva pagada al final de cada mes.
iv) La cantidad de deuda aún no pagada al final de cada mes.
v) El número de pagos mensuales necesarios para devolver el préstamo.
vi) La cuantía del último pago (ya que puede ser menor que A).

Comprobar en el programa utilizando los siguientes datos: P = \$40000; A = \$2000; i = 1% mensual.

k) Los estudiantes de una clase consiguieron las siguientes puntuaciones en seis exámenes de un curso de programación en C.

Nombre	Puntuaciones					
Adrián	45	80	80	95	55	75
Antonio	60	50	70	75	55	80
Carmen	40	30	10	45	60	55
Félix	0	5	5	0	10	5
Guillermo	90	85	100	95	90	90
José	95	90	80	95	85	80
Lidia	35	50	55	65	45	70
Maruja	75	60	75	60	70	80
Pedro	85	75	60	85	90	100
Raúl	50	60	50	35	65	70
Rosa	70	60	75	70	55	75
Sonia	10	25	35	20	30	10
Víctor	25	40	65	75	85	95
Zoraida	65	80	70	100	60	95

Escribir un programa interactivo en C que acepte como entrada cada nombre de estudiante y sus puntuaciones, determine la nota media de cada estudiante y escriba a continuación el nombre del estudiante, las notas de los exámenes y la media calculada.

l) Modificar el programa escrito en el apartado *k*) para que calcule la media dando distinto peso a cada examen. Suponer, por ejemplo, que cada uno de los primeros cuatro exámenes contribuye con el 15 por ciento de la nota final, y que cada uno de los dos últimos contribuye con el 20 por ciento.

m) Ampliar el programa escrito en el apartado anterior para que calcule además la nota media de la clase.

n) Escribir un programa en C que permita que se pueda utilizar la computadora como una calculadora normal. Considerar sólo las operaciones aritméticas básicas (suma, resta, multiplicación y división). Incluir una memoria en la que se pueda almacenar un número.

o) Generar la siguiente «pirámide» de dígitos utilizando bucles anidados.

```
         1
        232
       34543
      4567654
     567898765
    67890109876
   7890123210987
  890123454321098
 90123456765432109
0123456789876543210
```

No escribir simplemente diez cadenas de varios dígitos. Conseguir una fórmula para *generar* los dígitos correspondientes para cada línea.

p) Generar una gráfica de la función

$y = e^{-0.1t} \text{ sen } 0.5t$

en una impresora utilizando un asterisco (*) para cada punto que aparezca en la gráfica. Hacer que la gráfica se desarrolle verticalmente, con un punto (un asterisco) por línea. (*Sugerencia:* Cada línea impresa debe consistir en un asterisco precedido por el número adecuado de espacios en blanco. Determinar la posición del asterisco redondeando el valor de y al entero más cercano y utilizando una escala adecuada al número de caracteres por línea.)

q) Escribir un programa interactivo en C que pase una cantidad entera positiva a numeración romana (por ejemplo 12 se convierte en XII, 14 en XIV, etc.). Diseñar el programa para que se ejecute repetidamente, hasta que se introduzca cero.

r) Escribir un programa interactivo en C que convierta una fecha introducida de la forma `mm-dd-aa` (por ejemplo: 4-12-99) en un entero que indique el número de días a partir del 1 de enero de 1980. Si el año no es superior a 1999 (es decir, si `aa` ≤ 99), se pueden utilizar las siguientes relaciones:

i) El día del año en curso se puede determinar aproximadamente mediante

```
dia = (int) (30.42 * (mm -1)) + dd
```

ii) Si `mm == 2` (febrero), *incrementar* el valor de `dia` en 1.
iii) Si `mm > 2` y `mm < 8` (marzo, abril, mayo, junio o julio), *decrementar* el valor de `dia` en 1.
iv) Si `aa % 4 == 0` y `mm > 2` (año bisiesto), *incrementar* el valor de `dia` en 1.
v) *Incrementar* el valor de `dia` en 1461 por cada ciclo completo de cuatro años a partir del 1-1-80.
vi) *Incrementar* `dia` en 365 por cada año completo a partir del último ciclo completo de cuatro años, y sumar entonces 1 (por el año bisiesto más reciente).

Comprobar el programa con la fecha actual, o cualquier otro día a elegir.

s) Ampliar el apartado *r*) para permitir años del calendario por encima del año 1999 (el Ejemplo 10.28 presenta una solución para una versión más avanzada de este problema).

CAPÍTULO 7

Funciones

Ya hemos visto que en C se utilizan funciones de biblioteca con el fin de realizar un cierto número de operaciones o cálculos de uso común (ver sección 3.6). Sin embargo, C permite también al programador definir sus propias funciones que realicen determinadas tareas. Este capítulo se centra en la creación y utilización de estas funciones definidas por el programador.

El uso de funciones definidas por el programador permite dividir un programa grande en un cierto número de componentes más pequeñas, teniendo cada una de ellas un propósito único e identificable. Por tanto, un programa en C se puede *modularizar* mediante el uso inteligente de las funciones. (C no incluye otras formas de desarrollo modular de programas, como los procedimientos en Pascal o las subrutinas en Fortran.)

Hay varias ventajas de esta forma de desarrollo modular de los programas. Por ejemplo, muchos programas requieren que se acceda repetidamente a un grupo determinado de instrucciones desde varias partes distintas del programa. Las instrucciones repetidas se pueden incluir dentro de una sola función, a la que se puede acceder cuando sea necesario. Además, se puede transferir un conjunto de datos diferente a la función cada vez que se accede a ella. Por tanto, *el uso de funciones evita la necesidad de repetir las mismas instrucciones de forma redundante.*

Igualmente importante es la *claridad lógica* resultante de la descomposición de un programa en varias funciones concisas, representando cada función alguna parte bien definida del problema global. Estos programas son más fáciles de escribir y de depurar, y su estructura lógica es más fácil de entender que la de aquellos programas que carecen de este tipo de estructuras. Esto es especialmente cierto en programas grandes y complicados. La mayoría de los programas en C se modularizan de esta manera, aun cuando no impliquen la ejecución repetida de las mismas tareas. De hecho, la descomposición de un programa en módulos individuales se considera generalmente parte importante de la buena práctica de la programación.

La utilización de funciones permite también al programador construir una *biblioteca a medida* de rutinas de uso frecuente o de rutinas que se ocupen del manejo de elementos dependientes del sistema. Cada rutina se puede programar como una función por separado y almacenar en un archivo de biblioteca especial. Si un programa requiere una determinada rutina, se puede añadir la correspondiente función de biblioteca al programa durante el proceso de compilación. Por tanto, muchos programas distintos pueden utilizar una misma función. Esto evita la reescritura del código de las funciones. Además favorece la *portabilidad*, ya que se pueden escribir programas sin prestar atención a las características dependientes del sistema.

En este capítulo veremos cómo se definen las funciones y cómo se puede acceder a ellas desde diferentes partes de un programa en C. Veremos también cómo se puede pasar información a una función. Trataremos la utilización de *prototipos de funciones*, recomendada por el estándar

ANSI actual. Y finalmente, discutiremos una importante e interesante técnica de programación conocida como *recursividad*, en la que una función puede acceder a sí misma de forma repetida.

7.1. INTRODUCCIÓN

Una *función* es un segmento de programa que realiza determinadas tareas bien definidas. Todo programa en C consta de una o más funciones (ver sección 1.5). Una de estas funciones tiene que llamarse `main`. La ejecución del programa siempre comenzará por las instrucciones contenidas en `main`. Se pueden subordinar funciones adicionales a `main`, y posiblemente unas a otras.

Si un programa contiene varias funciones, sus definiciones pueden aparecer en cualquier orden, pero deben ser independientes unas de otras. Esto es, una definición de una función no puede estar incluida en otra.

Cuando se *accede* a una función desde alguna determinada parte del programa (cuando se «llama» a una función), se ejecutan las instrucciones de que consta. Se puede acceder a una misma función desde varios lugares distintos del programa. Una vez que se ha completado la ejecución de una función, se devuelve el control al punto desde el que se accedió a ella.

Generalmente, *una función procesará la información que le es pasada desde el punto del programa en donde se accede a ella y devolverá un solo valor*. La información se le pasa a la función mediante unos identificadores especiales llamados *argumentos* (también denominados *parámetros*) y es devuelta por medio de la instrucción `return`. Sin embargo, algunas funciones aceptan información pero no devuelven nada (por ejemplo, la función de biblioteca `printf`), mientras que otras funciones (la función de biblioteca `scanf`) devuelven varios valores.

EJEMPLO 7.1. Conversión de un carácter de minúscula a mayúscula. En el Ejemplo 3.31 vimos un sencillo programa en C que leía un solo carácter, lo convertía en mayúscula utilizando la función de biblioteca `toupper` y lo escribía. Consideremos ahora un programa similar, aunque definiremos nuestra propia función para realizar la conversión de minúscula a mayúscula.

Nuestro propósito al hacer esto es ilustrar los principales elementos involucrados en el uso de funciones. Por tanto, el lector debe concentrarse en la lógica general del programa y no preocuparse de los detalles de cada instrucción concreta.

He aquí el programa completo.

```
/* convertir un carácter en minúscula a mayúscula
   utilizando una función definida por el programador */

#include <stdio.h>

char minusc_a_mayusc(char c1)      /* definición de la función */

{
  char c2;

  c2 = (c1 >= 'a' && c1 <= 'z') ? ('A' + c1 - 'a') : c1;
  return(c2);

}
```

```
main()

{
   char minusc, mayusc;

   printf("Por favor, introduce una letra minúscula: ");
   scanf("%c", &minusc);
   mayusc = minusc_a_mayusc(minusc);
   printf("\nLa mayúscula equivalente es %c\n\n", mayusc);
}
```

Este programa consta de dos funciones, la función `main` requerida y la función definida por el programador `minusc_a_mayusc`, que convierte una minúscula a mayúscula. Nótese que `minusc_a_mayusc` efectúa la transformación real del carácter. Esta función transforma únicamente las letras en minúsculas; el resto de los caracteres se devuelven intactos. A través del argumento `c1` se transfiere a la función una letra en minúscula, y se devuelve la mayúscula correspondiente, `c2`, a la parte del programa que hizo la llamada (`main`), mediante la instrucción `return`.

Consideremos ahora la función `main`, que está a continuación de la función `minusc_a_mayusc`. Esta función lee un carácter (que puede ser o no una letra en minúscula) y se lo asigna a la variable de tipo carácter `minusc`. Entonces `main` llama a `minusc_a_mayusc`, le transfiere el carácter en minúscula (`minusc`) y recibe el carácter equivalente en mayúscula (`mayusc`). Se escribe a continuación el carácter en mayúscula y finaliza el programa. Nótese que las variables `minusc` y `mayusc` en `main` se corresponden con las variables `c1` y `c2` dentro de `minusc_a_mayusc`, respectivamente.

En el resto del capítulo consideraremos las reglas asociadas con las definiciones de las funciones y los accesos a éstas.

7.2. DEFINICIÓN DE UNA FUNCIÓN

La definición de una función tiene dos componentes principales: la *primera línea* (incluyendo las *declaraciones de los argumentos*) y el *cuerpo* de la función.

La primera línea de la definición de una función contiene la especificación del tipo de valor devuelto por la función, seguido del nombre de la función y (opcionalmente) un conjunto de argumentos, separados por comas y encerrados entre paréntesis. Cada argumento viene precedido por su declaración de tipo. Si la definición de la función no incluye ningún argumento hay que incluir detrás del nombre de la función un par de paréntesis vacíos.

En términos generales, la primera línea se puede escribir así:

`tipo-de-dato nombre(tipo 1 arg 1, tipo 2 arg 2, . . ., tipo n arg n)`

en donde *`tipo-de-dato`* representa el tipo de datos del valor que devuelve la función y *`nombre`* el nombre de la función, y *`tipo 1`*, *`tipo 2`*, ..., *`tipo n`* representan los tipos de datos de los argumentos *`arg 1`*, *`arg 2`*, ..., *`arg n`*. Los tipos de datos se suponen enteros cuando no se indican explícitamente. Sin embargo, la omisión de los tipos de datos se considera una mala práctica de programación, aun cuando éstos sean enteros.

Los argumentos se denominan *argumentos formales*, ya que representan los nombres de los elementos que se transfieren a la función desde la parte del programa que hace la llamada. Tam-

bién se llaman *parámetros* o *parámetros formales*. (Los argumentos correspondientes en la *referencia* a la función se denominan *argumentos reales*, ya que definen la información que realmente se transfiere. Algunos autores llaman *argumentos* simplemente a los argumentos reales, o *parámetros reales*.) Los identificadores utilizados como argumentos formales son «locales», en el sentido de que no son reconocidos fuera de la función. Por tanto, los nombres de los argumentos formales no tienen por qué coincidir con los nombres de los argumentos reales en la parte del programa que hace la llamada. Sin embargo, cada argumento formal debe ser del mismo *tipo de datos* que el dato que recibe desde el punto de llamada.

El resto de la definición de la función es una instrucción compuesta que define las acciones que debe realizar ésta. Se suele llamar a esta instrucción compuesta *cuerpo* de la función. Como cualquier otra instrucción compuesta, puede contener instrucciones de expresión, otras instrucciones compuestas, instrucciones de control, etc. Debe incluir una o más instrucciones `return` para devolver un valor al punto de llamada.

Una función puede acceder a otras funciones. De hecho, puede acceder a sí misma (este proceso se conoce como *recursividad* y se trata en la sección 7.6).

EJEMPLO 7.2. Consideremos la función `minusc_a_mayusc`, incluida en el Ejemplo 7.1.

```
char minusc_a_mayusc(char c1)        /* función de conversión
                                        definida por el programador */
{
   char c2;

   c2 = (c1 >= 'a' && c1 <= 'z') ? ('A' + c1 - 'a') : c1;
   return(c2);

}
```

La primera línea contiene el nombre de la función `minusc_a_mayusc`, seguido del argumento formal `c1`, que se encuentra entre paréntesis. El *nombre de función* viene precedido por el tipo de dato `char`, que describe el elemento devuelto por la función. Además, el *argumento formal* `c1` viene precedido por el tipo de dato `char`. Este último tipo de dato, que se incluye dentro del par de paréntesis, se refiere al argumento formal. El argumento formal, `c1`, representa el carácter en minúscula que se transfiere *a* la función desde el punto de llamada.

El cuerpo de la función comienza en la segunda línea, con la declaración de la variable local de tipo carácter `c2`. (Observe la diferencia entre el *argumento formal* `c1` y la *variable local* `c2`.) A continuación de la declaración de `c2` se encuentra una instrucción que comprueba si `c1` representa una letra minúscula y realiza entonces la conversión. Se devuelve el carácter original intacto si no es una letra minúscula. Finalmente, la instrucción `return` (ver a continuación) hace que se devuelva el carácter convertido al punto de llamada a la función.

Se devuelve información desde la función hasta el punto del programa desde donde se llamó mediante la instrucción `return`. La instrucción `return` también hace que se devuelva el control al punto de llamada.

En términos generales se puede escribir la instrucción `return` de la siguiente forma:

```
return expresión;
```

Se devuelve el valor de *expresión* al punto de llamada, como en el Ejemplo 7.2. La *expresión* es opcional. Si se omite la *expresión*, la instrucción `return` simplemente devuelve el control al punto del programa desde donde se llamó a la función, sin ninguna transferencia de información.

Sólo se puede incluir una expresión en la instrucción `return`. Por tanto, una función sólo puede devolver un valor al punto de llamada mediante la instrucción `return`.

Una definición de función puede incluir varias instrucciones `return`, conteniendo cada una de ellas una expresión distinta. Las funciones que incluyen varias bifurcaciones suelen requerir varias instrucciones `return`.

EJEMPLO 7.3. He aquí la función `minusc_a_mayusc` de los Ejemplos 7.1 y 7.2, con algunas variaciones.

```
char minusc_a_mayusc(char c1)     /* función de conversión
                                     definida por el programador */

{
   if (c1 >= 'a' && c1 <= 'z')
      return('A' + c1 - 'a');
   else
      return(c1);
}
```

Esta función utiliza la instrucción `if - else` en lugar del operador condicional. Es menos compacta que la versión anterior, pero puede resultar de más fácil comprensión. Por otra parte, esta función no necesita la variable local `c2`.

Nótese que la función contiene dos instrucciones `return`. La primera devuelve una expresión que representa la mayúscula correspondiente a la minúscula dada; la segunda devuelve el carácter original, sin cambios.

La instrucción `return` puede faltar en la definición de una función, aunque esto se considera generalmente como una mala práctica de programación. Si una función alcanza el final del bloque sin encontrarse una instrucción `return`, se devuelve el control al punto de llamada sin devolverse ninguna información. Se recomienda en estos casos una instrucción `return` vacía (sin expresión), para hacer más clara la lógica de la función y hacer más cómodas las modificaciones futuras de la función.

EJEMPLO 7.4. La siguiente función acepta dos cantidades enteras y determina el valor mayor, que se escribe a continuación. La función no devuelve ninguna información al punto de llamada.

```
maximo(int x, int y)        /* determinar el máximo de
                               dos cantidades enteras    */

{
   int z;

   z = (x >= y) ? x : y;
   printf("\n\nValor máximo = %d", z);
   return;

}
```

Observe que se ha incluido una instrucción `return` vacía como práctica de buena programación. De todas formas, la función actuaría igual si la instrucción `return` no se encontrara presente.

EJEMPLO 7.5. El *factorial* de un entero positivo se define como $n! = 1 \times 2 \times 3 \times \ldots \times n$. Por tanto, $2! = 1 \times 2 = 2$; $3! = 1 \times 2 \times 3 = 6$; $4! = 1 \times 2 \times 3 \times 4 = 24$; y así sucesivamente.

La función que se muestra a continuación calcula el factorial de un entero positivo dado, *n*. Se devuelve el factorial como un entero largo, ya que el factorial crece muy rápidamente al hacerlo *n*. (Por ejemplo, 8! = 40320. Este valor, expresado como un entero ordinario, puede ser muy grande para algunas computadoras.)

```
long int factorial(int n)      /* calcular el factorial de n */

{
   int i;
   long int prod = 1;

   if (n > 1)
      for (i = 2; i <= n; ++i)
           prod *= i;
   return(prod);
}
```

Observe la especificación de tipo `long int` que se incluye en la primera línea de la definición de la función. También se declara la variable local `prod` como entero largo dentro de la función. Observe que a `prod` se le asigna un valor inicial de `1`, aunque su valor se recalcula en el bucle `for`. El valor final de `prod`, que es devuelto por la función, representa el valor deseado del factorial de *n*.

Si el tipo de datos especificado en la primera línea es inconsistente con la expresión que aparece en la instrucción `return`, el compilador intentará convertir la cantidad representada por la expresión al tipo de datos especificado en la primera línea. El resultado de esto puede ser un error de compilación o una pérdida parcial de datos (por ejemplo, debido al truncamiento). En cualquier caso, se deben evitar inconsistencias de este tipo.

EJEMPLO 7.6. La siguiente definición de función es idéntica a la del Ejemplo 7.5, excepto en la primera línea, en la que no aparece especificación de tipo para el valor devuelto por la función.

```
factorial(int n)               /* calcular el factorial de n */

{
   int i;
   long int prod = 1;

   if (n > 1)
      for (i = 2; i <= n; ++i)
           prod *= i;
   return(prod);
}
```

La función devolverá una cantidad entera ordinaria, ya que no hay una declaración explícita de tipo de datos en la primera línea de la definición de la función. Sin embargo, la cantidad que se va a devolver (`prod`) se declara como entero largo dentro de la función. El resultado de esta inconsistencia puede ser un error. (Algunos compiladores generarán un mensaje de error y detendrán en ese punto la compilación.) Sin embargo, se puede evitar el problema añadiendo una declaración de tipo `long int` en la primera línea de la definición de la función, como en el Ejemplo 7.5.

Se puede utilizar la palabra reservada `void` como especificador de tipo cuando se define una función que no devuelve nada. La presencia de esta palabra reservada no es obligatoria, pero es una buena práctica de programación.

EJEMPLO 7.7. Consideremos de nuevo la función que aparece en el Ejemplo 7.4, que acepta dos cantidades enteras y escribe la mayor de las dos. Recordar que esta función no devuelve nada al punto de llamada. Por tanto, la función se puede escribir así:

```
void maximo(int x, int y)       /* determinar el máximo de
                                   dos cantidades enteras    */

{
   int z;

   z = (x >= y) ? x : y;
   printf("\n\nValor máximo = %d", z);
   return;
}
```

Esta función es idéntica a la incluida en el Ejemplo 7.4, excepto en que se ha añadido la palabra reservada `void` en la primera línea, indicando que la función no devuelve nada.

7.3. ACCESO A UNA FUNCIÓN

Se puede *acceder* (*llamar*) a una función especificando su nombre, seguido de una lista de argumentos encerrados entre paréntesis y separados por comas. Si la llamada a la función no requiere ningún argumento, se debe escribir a continuación del nombre de la función un par de paréntesis vacíos. La llamada a la función puede formar parte de una expresión simple (como por ejemplo una instrucción de asignación) o puede ser uno de los operandos de una expresión más compleja.

Los argumentos que aparecen en la llamada a la función se denominan *argumentos reales*, en contraste con los argumentos formales que aparecen en la primera línea de la definición de la función. (También se llaman simplemente argumentos, o parámetros reales.) En una llamada normal a una función, habrá un argumento real por cada argumento formal. Los argumentos reales pueden ser constantes, variables simples, o expresiones más complejas. No obstante, cada argumento real debe ser del mismo tipo de datos que el argumento formal correspondiente. Recordar que el *valor* de cada argumento real es transferido a la función y asignado al correspondiente argumento formal.

Si la función devuelve un valor, el acceso a la función se suele escribir a menudo como una instrucción de asignación; esto es,

```
y = polinomio(x);
```

Este acceso a la función hace que a la variable y se le asigne el valor devuelto por la función.

Por otro lado, si la función no devuelve nada, el acceso a la función se escribe por separado, como en,

```
visualizar(a, b, c);
```

Este acceso a la función hace que se procesen internamente (se visualicen) los valores de a, b y c dentro de la función.

EJEMPLO 7.8. Consideremos de nuevo el programa que presentamos originalmente en el Ejemplo 7.1, que lee un carácter y lo convierte de minúscula a mayúscula utilizando una función definida por el programador, y luego escribe el equivalente en mayúscula.

```
/* convertir un carácter en minúscula a mayúscula
   utilizando una función definida por el programador */

#include <stdio.h>

char minusc_a_mayusc(char c1)     /* definición de la función */

{
   char c2;

   c2 = (c1 >= 'a' && c1 <= 'z') ? ('A' + c1 - 'a') : c1;
   return(c2);
}

void main(void)

{
   char minusc, mayusc;

   printf("Por favor, introduce una letra minúscula: ");
   scanf("%c", &minusc);
   mayusc = minusc_a_mayusc(minusc);
   printf("\nLa mayúscula equivalente es %c\n\n", mayusc);
}
```

En este programa, main contiene sólo una llamada a la función definida por el programador minusc_a_mayusc. La llamada es una parte de la expresión de asignación mayusc = minusc_a_mayusc(minusc).

La llamada a la función contiene un argumento real, la variable de tipo carácter minusc. Observe que el argumento formal correspondiente, c1, dentro de la definición de la función es también una variable de tipo carácter.

Cuando se accede a la función, se transfiere el valor de `minusc` a la función. Este valor es representado por `c1` dentro de la función. A continuación se determina el valor de la mayúscula correspondiente, `c2`, y se devuelve al punto de llamada, en donde se asigna a la variable de tipo carácter `mayusc`.

Observe que se pueden combinar las dos últimas instrucciones de `main` de la forma siguiente:

```
printf("\nLa mayúscula equivalente es %c\n\n", minusc_a_mayusc(minusc));
```

Ahora la llamada a `minusc_a_mayusc` es un argumento real de la función de biblioteca `printf`. Observe también que ya no se necesita la variable `mayusc`.

Finalmente, observe la forma en que se escribe la primera línea de `main`, esto es, `void main(void)`. Esto está permitido en el estándar ANSI, aunque algunos compiladores no acepten `void` como tipo de retorno. En consecuencia, muchos autores (y muchos programadores) escriben la primera línea de `main` como `main(void)` o simplemente `main()`. Usaremos esta última opción en el resto del libro.

Puede haber diversas llamadas a la misma función desde varios lugares de un programa. Los argumentos reales pueden ser distintos de una llamada a otra. En todo caso, dentro de cada llamada a una función *los argumentos reales deben corresponderse con los argumentos formales de la definición de la función; es decir, el número de argumentos reales debe ser el mismo que el número de argumentos formales y cada argumento real debe ser del mismo tipo de datos que el correspondiente argumento formal.*

EJEMPLO 7.9. Mayor de tres cantidades enteras. El siguiente programa determina la mayor de tres cantidades enteras. Este programa utiliza una función que determina la mayor de dos cantidades enteras. La función es semejante a la definida en el Ejemplo 7.4, salvo que la función de este ejemplo devuelve el valor mayor al punto de llamada en lugar de visualizarlo.

La estrategia global es determinar la mayor de las dos primeras cantidades y compararla a continuación con la tercera. La cantidad mayor se escribe en la parte principal del programa.

```
/* determinar la mayor de tres cantidades enteras */

#include <stdio.h>

int maximo(int x, int y)          /* determinar la mayor de dos
                                     cantidades enteras */

{
   int z;

   z = (x >= y) ? x : y;
   return(z);
}

main()

{
   int a, b, c, d;

   /* leer las cantidades enteras */
   printf("\na = ");
   scanf("%d", &a);
   printf("\nb = ");
```

```
    scanf("%d", &b);
    printf("\nc = ");
    scanf("%d", &c);

    /* calcular y visualizar el valor máximo */

    d = maximo(a, b);
    printf("\n\nmáximo = %d", maximo(c, d));
}
```

Se accede a la función `maximo` desde dos lugares diferentes en `main`. En la primera llamada a `maximo`, los argumentos reales son las variables `a` y `b`, mientras que en la segunda llamada los argumentos son `c` y `d` (`d` es una variable temporal que representa el valor máximo de `a` y `b`).

Observe que las dos instrucciones que acceden a `maximo`, es decir,

```
d = maximo(a, b);
printf("\n\nmáximo = %d", maximo(c, d));
```

se pueden reemplazar por la instrucción

```
printf("\n\nmáximo = %d", maximo(c, maximo(a, b)));
```

En esta instrucción vemos que una de las llamadas a `maximo` es un argumento para la otra llamada. Es decir, pueden incluirse las llamadas, una dentro de otra, y no se necesita la variable intermedia `d`. Se permite tener llamadas a funciones incluidas dentro de otras, aunque en algunos casos la lógica subyacente puede resultar menos clara. Por tanto, en general deberían ser evitadas por los programadores que se están iniciando.

7.4. PROTOTIPOS DE FUNCIONES

En los programas de este capítulo examinados con anterioridad, la función definida por el programador siempre ha *precedido* a `main`. De este modo, al compilar el programa, la función definida por el programador estaría definida antes del primer acceso a la función. Sin embargo, muchos programadores prefieren un enfoque «descendente», en el que `main` aparece antes de la definición de función definida por el programador. En tales situaciones el acceso a la función (dentro de `main`) precedería la definición de la función. Esto puede confundir al compilador, a menos que se alerte primero al compilador del hecho de que la función a acceder se definirá más adelante en el programa. Para esta finalidad se utilizan los *prototipos de funciones*.

Los prototipos de funciones normalmente se escriben al comienzo del programa, delante de todas las funciones definidas por el programador (incluida `main`). La forma general de un prototipo de función es:

`tipo-de-datos nombre(tipo 1 arg 1, tipo 2 arg2, . . ., tipo n arg n);`

en donde *`tipo-de-datos`* representa el tipo de datos del elemento devuelto por la función, *`nombre`* representa el nombre de la función, y *`tipo 1`*, *`tipo 2`*, ..., *`tipo n`* representan los tipos de datos de los argumentos *`arg 1`*, *`arg 2`*, ..., *`arg n`*. Observe que un prototipo de

función es similar a la primera línea de una definición de función (si bien un prototipo de función finaliza con un punto y coma).

No es necesario declarar en ningún lugar del programa los nombres de los argumentos de un prototipo de función, puesto que éstos son nombres de argumentos «ficticios» que sólo se reconocen dentro del prototipo. De hecho se pueden omitir los nombres de los argumentos (aunque no es una buena idea hacerlo); sin embargo, los *tipos de datos* de los argumentos son esenciales.

En la práctica, normalmente se incluyen los nombres de los argumentos y éstos frecuentemente coinciden con los nombres de los argumentos reales que aparecen en alguna de las llamadas a la función. Los tipos de datos de los argumentos reales se deben adecuar a los tipos de datos de los argumentos en el prototipo.

Los prototipos de funciones no son obligatorios en C. Sin embargo, son aconsejables, ya que facilitan la comprobación de errores ulterior entre las llamadas a una función y la definición de función correspondiente.

EJEMPLO 7.10. Cálculo de factoriales. He aquí un programa completo en C que calcula el factorial de una cantidad entera positiva. El programa utiliza la función `factorial`, definida en el Ejemplo 7.5. Observe que la definición de la función precede a `main`, al igual que en los anteriores ejemplos de programas de este capítulo.

```
/* calcular el factorial de una cantidad entera */

#include <stdio.h>

long int factorial(int n)

/* calcular el factorial de n */

{
   int i;
   long int prod = 1;

   if (n > 1)
      for (i = 2; i <= n; ++i)
          prod *= i;
   return(prod);
}

main()

{
   int n;

   /* leer la cantidad entera */

   printf("\nn = ");
   scanf("%d", &n);

   /* calcular y visualizar el factorial */

   printf("\nn! = %ld", factorial(n));

}
```

La función definida por el programador (`factorial`) utiliza un argumento entero (`n`) y dos variables locales —un entero ordinario (`i`) y un entero largo (`prod`)—. Puesto que la función devuelve un entero largo, la declaración de tipo `long int` aparece en la primera línea de la definición de la función.

A continuación se presenta otra versión del programa, escrita descendentemente (es decir, `main` aparece delante de `factorial`). Observe la presencia del prototipo de la función al comienzo del programa. El prototipo de la función indica que se definirá, posteriormente en el programa, una función llamada `factorial`, que acepta una cantidad entera y devuelve un entero largo.

```
/* calcular el factorial de una cantidad entera */

#include <stdio.h>

long int factorial(int n);    /* prototipo de función */

main()

{
   int n;

   /* leer la cantidad entera */

   printf("\nn = ");
   scanf("%d", &n);

   /* calcular y visualizar el factorial */

   printf("\nn! = %ld", factorial(n));

}

long int factorial(int n)

/* calcular el factorial de n */

{
   int i;
   long int prod = 1;

   if (n > 1)
      for (i = 2; i <= n; ++i)
          prod *= i;
   return(prod);
}
```

Las llamadas a las funciones pueden abarcar varios niveles en un programa. Esto es, la función `A` puede llamar a la función `B`, la cual puede llamar a la función `C`, etc. También, la función `A` puede llamar directamente a la función `C`, y así sucesivamente.

EJEMPLO 7.11. Simulación de un juego de azar (juego de dados «Craps»). Éste es un interesante problema de programación que incluye varias llamadas a funciones a diferentes niveles. Se requieren funciones de biblioteca y funciones definidas por el programador.

Existe un popular juego de dados llamado «Craps» en el que se lanzan los dados una vez o más hasta que se gana o se pierde. El juego se puede simular en una computadora sustituyendo el lanzamiento real de los dados con la generación de números aleatorios.

Hay dos formas de ganar. Puede lanzar los dados y obtener una puntuación de 7 u 11; o puede obtener 4, 5, 6, 8, 9 o 10 en la primera tirada y conseguir de nuevo la misma puntuación en alguna de las siguientes tiradas antes de obtener un 7. Existen también dos formas de perder. Puede lanzar los dados una vez y obtener 2, 3 o 12; o puede obtener 4, 5, 6, 8, 9 o 10 en la primera tirada y obtener un 7 en una tirada posterior antes de repetir la puntuación original.

Desarrollaremos el juego interactivamente, de forma que se simule una tirada de dados cada vez que se pulse la tecla Intro. Aparecerá entonces un mensaje informando del resultado de cada lanzamiento. Al final de cada juego se preguntará si se desea continuar jugando o no.

Nuestro programa requerirá un generador de números aleatorios que produzca enteros distribuidos uniformemente entre 1 y 6. (Por *uniformemente distribuidos* entendemos que cada entero entre 1 y 6 tiene la misma probabilidad de aparecer.) La mayoría de las versiones de C incluyen un generador de números aleatorios entre sus funciones de biblioteca. Estos generadores de números aleatorios suelen devolver típicamente un número en coma flotante uniformemente distribuido entre 0 y 1, o una cantidad entera uniformemente distribuida entre 0 y algún valor entero muy grande.

Utilizaremos una función de biblioteca llamada rand, que devuelve un entero uniformemente distribuido entre 0 y $2^{15}-1$ (esto es, entre 0 y 32767). Convertiremos entonces cada cantidad entera aleatoria en un número en coma flotante x, que se encontrará entre 0 y 0.99999.... Para hacer esto escribimos

```
x = rand() / 32768.0
```

Observe que el denominador se ha escrito como constante en coma flotante. Esto hace que el cociente, y por tanto x, sea en coma flotante.

La expresión

```
(int) (6 * x)
```

será un entero truncado, con valor uniformemente distribuido entre 0 y 5. Por tanto, obtenemos el resultado deseado simplemente añadiendo 1; esto es,

```
n = 1 + (int) (6 * x)
```

Este valor representará el resultado del lanzamiento de un dado. Si repetimos este proceso una segunda vez y sumamos los resultados, obtenemos el resultado de lanzar dos dados.

La siguiente función utiliza la estrategia descrita anteriormente para simular la tirada de un par de dados.

```
int tirada(void) /* simula el lanzamiento de un par de dados */

{
  float x1, x2;  /* números en coma flotante aleatorios entre 0 y 1 */
  int n1, n2;    /* enteros aleatorios entre 1 y 6 */

  x1 = rand() / 32768.0;
  x2 = rand() / 32768.0;
```

```
    n1 = 1 + (int) (6 * x1);/* simula primer dado */
    n2 = 1 + (int) (6 * x2);/* simula segundo dado */

    return(n1 + n2);   /* la puntuación es la suma de los dos dados */
}
```

La función devuelve el resultado de cada tirada (una cantidad entera con valor entre `2` y `12`). Observe que este resultado final *no* estará uniformemente distribuido, aunque los valores de `n1` y `n2` lo estén.

Definamos ahora otra función llamada `juego`, que simula una jugada completa. Por tanto, los dados se lanzarán tantas veces como sea necesario para determinar si se pierde o se gana. Esta función accederá a `tirada`. Todas las reglas del juego están presentes en esta función.

Podemos escribir el pseudocódigo de `juego` como sigue.

```
void juego(void)    /* simular una jugada completa */

{
   int puntos1, puntos2;

   /* informar al usuario de cómo tirar los dados */

   /* inicializar el generador de números aleatorios */

   puntos1 = tirada();

   switch (puntos1) {

   case 7:
   case 11:
        /* escribir un mensaje indicando que se ha ganado en la pri-
           mera tirada */

   case 2:
   case 3:
   case 12:

        /* escribir un mensaje indicando que se ha perdido en la
           primera tirada */

   case 4:
   case 5:
   case 6:
   case 8:
   case 9:
   case 10:

        do {
             /* instruir al usuario de cómo lanzar los dados de nuevo */

             puntos2 = tirada();

        } while (puntos2 != puntos1 && puntos2 != 7);

        if (puntos2 == puntos1)

           /* escribir un mensaje indicando que se ha ganado */
```

```
        else
            /* escribir un mensaje indicando que se ha perdido */
    }
    return;
}
```

La rutina `main` controlará la ejecución del juego. Esta rutina constará de un bucle `while` que contendrá la entrada/salida interactiva y la llamada a `juego`. Por tanto, podemos escribir el pseudocódigo de `main` como sigue.

```
main()
{
    /* declaraciones */

    /* inicializar el generador de números aleatorios */

    /* generar un mensaje de bienvenida */

    while ( /* el jugador desea seguir jugando */ ) {

        jugar();

        /* preguntar al jugador si desea continuar */
    }

    /* generar mensaje de despedida */

}
```

La función de biblioteca `srand` se utiliza para inicializar el generador de números aleatorios. Esta función requiere un entero positivo, llamado `semilla`, que establece la secuencia de números aleatorios generados por `rand`. Se generará una secuencia de números aleatorios diferente para cada semilla. Por comodidad, podemos incluir un valor de la semilla como constante simbólica dentro del programa. (Si el programa se ejecuta repetidamente con la misma semilla, se generará cada vez la misma secuencia de números aleatorios. Esto puede servir de ayuda al depurar el programa.)

He aquí el programa completo en C.

```
/* simulación del juego de dados "Craps" */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

#define SEMILLA 12345

void juego(void); /* prototipo de función */
int tirada(void); /* prototipo de función */

main()
```

```
{
  char respuesta = 'S';

  printf("Bienvenido al juego CRAPS\n\n");
  printf("Para lanzar los dados, pulsa Intro\n\n");

  srand(SEMILLA); /* inicializa el generador de números aleatorios */

  /* bucle principal */

  while (toupper(respuesta) != 'N') {
      juego();
      printf("\n¿Deseas jugar de nuevo? (S/N) ");
      scanf(" %c", &respuesta);
      printf("\n");
  }
  printf("Adiós, que lo pases bien");
}

void juego(void)    /* simular una jugada completa */
{
  int puntos1, puntos2;
  char nada;

  printf("\nPor favor lanza los dados . . .");
  scanf("%c", &nada);
  printf("\n");
  puntos1 = tirada();
  printf("\n%2d", puntos1);

  switch (puntos1) {

  case 7:  /* se ha ganado en la primera tirada */
  case 11:

        printf(" - ¡Felicidades! GANASTE en la primera tirada\n");
        break;

  case 2:  /* se ha perdido en la primera tirada */
  case 3:
  case 12:

        printf(" - ¡Lo siento! - PERDISTE en la primera tirada\n");
        break;

  case 4:  /* se requieren otras tiradas */
  case 5:
  case 6:
  case 8:
  case 9:
  case 10:
```

```
        do {
                printf(" - Lanza los dados de nuevo . . .");
                scanf("%c", &nada);
                puntos2 = tirada();
                printf("\n%2d", puntos2);
        } while (puntos2 != puntos1 && puntos2 != 7);

        if (puntos2 == puntos1)
           printf(" - GANAS por igualar tu primera puntuación\n");
        else
           printf(" - PIERDES por no igualar tu primera puntuación\n");
        break;
    }

    return;
}

int tirada(void)  /* simula el lanzamiento de un par de dados */

{
    float x1, x2;   /* números en coma flotante aleatorios entre 0 y 1 */
    int n1, n2;

    x1 = rand() / 32768.0;
    x2 = rand() / 32768.0;

    n1 = 1 + (int) (6 * x1); /* simula primer dado */
    n2 = 1 + (int) (6 * x2); /* simula segundo dado */

    return(n1 + n2); /* la puntuación es la suma de los dos dados */
}
```

Observe que `main` llama a `srand` y `juego`. Se le pasa un argumento a `srand` (el valor de `semilla`), pero no se le pasa ningún argumento a `juego`. Observe también que `juego` llama a `tirada` desde dos lugares diferentes, y `tirada` llama a `rand` desde dos lugares diferentes. No se pasan argumentos desde `juego` a `tirada`, ni desde `tirada` a `rand`. Sin embargo, `rand` devuelve un entero aleatorio a `tirada`, y `tirada` devuelve el valor de una expresión entera (el resultado de una tirada de los dados) a `juego`. Nótese que `juego` no devuelve ninguna información a `main`.

Dentro de `juego` hay dos referencias a la función `scanf`, cada una de las cuales lee un valor de la variable `nada`. Se debe entender que `nada` no se utiliza realmente en el programa. La presencia de la función `scanf` tiene la única finalidad de generar una espera en la ejecución del programa hasta que se pulse la tecla `Intro` (para simular una nueva tirada de los dados).

Este programa debe ejecutarse en un entorno interactivo, tal como una computadora personal. Se muestra a continuación una típica sesión interactiva. Por claridad, se han subrayado las respuestas del usuario.

```
Bienvenido al juego CRAPS

Para lanzar los dados, pulsa Intro (Intro)


Por favor lanza los dados . . .
```

```
 6 - Lanza los dados de nuevo . . .
10 - Lanza los dados de nuevo . . .
 7 - PIERDES por no igualar tu primera puntuación
¿Deseas jugar de nuevo? (S/N) s

Por favor lanza los dados . . .
 7 - ¡Felicidades! GANASTE en la primera tirada
¿Deseas jugar de nuevo? (S/N) s

Por favor lanza los dados . . .
11 - ¡Felicidades! GANASTE en la primera tirada
¿Deseas jugar de nuevo? (S/N) s

Por favor lanza los dados . . .
 8 - Lanza los dados de nuevo . . .
 5 - Lanza los dados de nuevo . . .
 7 - PIERDES por no igualar tu primera puntuación
¿Deseas jugar de nuevo? (S/N) s

Por favor lanza los dados . . .
 6 - Lanza los dados de nuevo . . .
 4 - Lanza los dados de nuevo . . .
 6 - GANAS por igualar tu primera puntuación
¿Deseas jugar de nuevo? (S/N) s

Por favor lanza los dados . . .
 3 - ¡Lo siento! - PERDISTE en la primera tirada
¿Deseas jugar de nuevo? (S/N) n
Adiós, que lo pases bien
```

7.5. PASO DE ARGUMENTOS A UNA FUNCIÓN

Cuando se le pasa un valor simple a una función mediante un argumento real, se *copia* el valor del argumento real a la función. Por tanto, *se puede modificar el valor del argumento formal dentro de la función, pero el valor del argumento real en la rutina que efectúa la llamada no cambiará*. Este procedimiento para pasar el valor de un argumento a una función se denomina *paso por valor*.

EJEMPLO 7.12. He aquí un sencillo programa en C que contiene una función que modifica el valor de su argumento.

```
#include <stdio.h>

void modificar(int a);    /* prototipo de función */

main()
{
  int a = 2;

  printf("\na = %d (desde main, antes de llamar a la función)", a);
  modificar(a);
  printf("\n\na = %d (desde main, después de llamar a la función)", a);
}

void modificar(int a)
{
  a *= 3;
  printf("\n\na = %d (desde la función, modificando valor)", a);
  return;
}
```

Se visualiza el valor original de a (a = 2) cuando comienza la ejecución de `main`. Este valor se pasa a la función `modificar`, en donde se multiplica por 3 y se visualiza el nuevo valor. Observe que es el *valor alterado* del argumento formal el que se visualiza en la función. Finalmente, el valor de a en `main` (el argumento real) se vuelve a visualizar, después de haberse devuelto el control a `main` desde `modificar`.

Cuando se ejecuta el programa, se genera la siguiente salida:

```
a = 2 (desde main, antes de llamar a la función)

a = 6 (desde la función, modificando el valor)

a = 2 (desde main, después de llamar a la función)
```

Estos resultados muestran que a no se ha modificado dentro de `main`, aunque se haya modificado el valor correspondiente de a en `modificar`.

Pasar un argumento por valor tiene sus ventajas e inconvenientes. Algo positivo es que permite que pueda proporcionarse como argumento real una expresión en lugar de necesariamente

una variable. Es más, si el argumento real es una simple variable, se protege su valor de posibles alteraciones por parte de la función. Por otra parte, impide que se transfiera información desde la función hasta el punto de llamada mediante los argumentos. Por tanto, *el paso por valor implica que la transferencia de información sólo pueda realizarse en un sentido.*

EJEMPLO 7.13. Cálculo de depreciación. Consideremos una variación del programa del Ejemplo 6.26. El objetivo final es calcular la depreciación en función del tiempo utilizando uno de los tres métodos comúnmente usados. Reescribiremos ahora el programa para que se utilice una función por separado para cada método. Esto nos permite organizar los componentes lógicos del programa de una forma más clara. Además, pasaremos un bloque de instrucciones repetidas de salida a una función, eliminando así una cierta programación redundante de la versión original del programa.

También aumentaremos la generalidad del programa de algún modo, ya que permitiremos que se puedan hacer distintos cálculos de la depreciación con los mismos datos de entrada. Se preguntará al usuario al final de cada conjunto de cálculos si desea que se vuelva a realizar otro nuevo conjunto de cálculos. Si la respuesta es sí, se le preguntará al usuario si desea introducir nuevos datos o no.

He aquí la nueva versión del programa, escrita descendentemente.

```
/* calcular la depreciación utilizando uno de tres métodos diferentes */

#include <stdio.h>
#include <ctype.h>

void lr(float val, int n);                    /* prototipo de función */
void bdd(float val, int n);                   /* prototipo de función */
void sda(float val, int n);                   /* prototipo de función */
void escribir_salida(int anual, float depreciacion, float valor);
                                              /* prototipo de función */
main()

{
   int n, eleccion = 0;
   float val;
   char resp1 = 'S', resp2 = 'S';

   while (toupper(resp1) != 'N') {

         /* leer datos de entrada */

         if (toupper(resp2) != 'N') {
            printf("\nValor original: ");
            scanf("%f", &val);
            printf("Número de años: ");
            scanf("%d", &n);
         }
         printf("\nMétodo: (1-LR  2-BDD  3-SDA) ");
         scanf("%d", &eleccion);

         switch (eleccion) {
```

```
        case 1:  /* método de línea recta */

            printf("\nMétodo de línea recta\n\n");
            lr(val, n);
            break;

        case 2:  /* método de balance doblemente declinante */

            printf("\nMétodo de balance doblemente declinante\n\n");
            bdd(val, n);
            break;

        case 3:  /* método de la suma de los dígitos de los años */

            printf("\nMétodo de la suma de los dígitos de los años\n\n");
            sda(val, n);
        }

        printf("\n\n¿Más cálculos? (S/N) ");
        scanf("%1s", &resp1);
        if (toupper(resp1) != 'N') {
            printf("¿Introducir un nuevo conjunto de datos? (S/N) ");
            scanf("%1s", &resp2);
        }
    }

    printf("\nHasta luego y que tenga un buen día\n");

}

void lr(float val, int n)  /* método de línea recta */

{
    float deprec;
    int anual;

    deprec = val/n;
    for (anual = 1; anual <= n; ++anual) {
        val -= deprec;
        escribir_salida(anual, deprec, val);
    }
    return;
}

void bdd(float val, int n) /* método de balance doblemente declinante */

{
    float deprec;
    int anual;
```

```
    for (anual = 1; anual <= n; ++anual) {
        deprec = 2*val/n;
        val -= deprec;
        escribir_salida(anual, deprec, val);
    }
    return;
}

void sda(float val, int n) /* método de la suma de los dígitos de
                                los años */

{
    float aux, deprec;
    int anual;

    aux = val;
    for (anual = 1; anual <= n; ++anual) {
        deprec = (n-anual+1)*aux / (n*(n+1)/2);
        val -= deprec;
        escribir_salida(anual, deprec, val);
    }
    return;
}

void escribir_salida(int anual, float depreciacion, float valor)
                                /* escribir los datos de salida */

{
    printf("Fin de año %2d", anual);
    printf("  Depreciación: %7.2f", depreciacion);
    printf("  Valor actual: %8.2f\n", valor);
    return;
}
```

Observe que aún se emplea la instrucción `switch`, como en el Ejemplo 6.26, aunque ahora hay sólo tres opciones en lugar de cuatro. (La opción cuarta, que finalizaba la ejecución en la versión anterior, se maneja ahora mediante el diálogo interactivo al final de los cálculos.) Se proporciona ahora una función por separado para cada tipo de cálculos. En particular, los cálculos del método de línea recta se efectúan dentro de la función `lr`, los del método del balance doblemente declinante dentro de `bdd`, y los de la suma de los dígitos de los años dentro de `sda`. Cada una de estas funciones incluye los argumentos formales `val` y `n`, que representan el valor original del objeto y su tiempo de vida, respectivamente. Observe que el valor de `val` es alterado dentro de cada función, aunque el valor original asignado a `val` permanece sin alteración dentro de `main`. Es esto lo que permite repetir el conjunto de cálculos con el mismo valor de entrada.

La última función, `escribir_salida`, hace que los resultados de cada conjunto de cálculos se escriban año a año. Se accede a esta función desde `lr`, `bdd` y `sda`. En cada llamada a `escribir_salida`, el valor *modificado* de `val` se transfiere como argumento real, junto con el año en curso (`anual`) y la depreciación del año en curso (`deprec`). Observe que estas cantidades son llamadas `valor`, `anual` y `depreciación`, respectivamente, dentro de `escribir_salida`.

A continuación se muestra un ejemplo de sesión interactiva con este programa.

```
Valor original: 8000
Número de años: 10

Método: (1-LR 2-BDD 3-SDA) 1

Método de línea recta

Fin de año  1  Depreciación:  800.00  Valor actual:  7200.00
Fin de año  2  Depreciación:  800.00  Valor actual:  6400.00
Fin de año  3  Depreciación:  800.00  Valor actual:  5600.00
Fin de año  4  Depreciación:  800.00  Valor actual:  4800.00
Fin de año  5  Depreciación:  800.00  Valor actual:  4000.00
Fin de año  6  Depreciación:  800.00  Valor actual:  3200.00
Fin de año  7  Depreciación:  800.00  Valor actual:  2400.00
Fin de año  8  Depreciación:  800.00  Valor actual:  1600.00
Fin de año  9  Depreciación:  800.00  Valor actual:   800.00
Fin de año 10  Depreciación:  800.00  Valor actual:     0.00

¿Más cálculos? (S/N) s
¿Introducir un nuevo conjunto de datos? n

Método: (1-LR 2-BDD 3-SDA) 2

Método de balance doblemente declinante

Fin de año  1  Depreciación: 1600.00  Valor actual:  6400.00
Fin de año  2  Depreciación: 1280.00  Valor actual:  5120.00
Fin de año  3  Depreciación: 1024.00  Valor actual:  4096.00
Fin de año  4  Depreciación:  819.20  Valor actual:  3276.80
Fin de año  5  Depreciación:  655.36  Valor actual:  2621.44
Fin de año  6  Depreciación:  524.29  Valor actual:  2097.15
Fin de año  7  Depreciación:  419.43  Valor actual:  1677.72
Fin de año  8  Depreciación:  335.54  Valor actual:  1342.18
Fin de año  9  Depreciación:  268.44  Valor actual:  1073.74
Fin de año 10  Depreciación:  214.75  Valor actual:   858.99

¿Más cálculos? (S/N) s
¿Introducir un nuevo conjunto de datos? n

Método: (1-LR 2-BDD 3-SDA) 3

Método de la suma de los dígitos de los años

Fin de año  1  Depreciación: 1454.55  Valor actual:  6545.45
Fin de año  2  Depreciación: 1309.09  Valor actual:  5236.36
Fin de año  3  Depreciación: 1163.64  Valor actual:  4072.73
Fin de año  4  Depreciación: 1018.18  Valor actual:  3054.55
Fin de año  5  Depreciación:  872.73  Valor actual:  2181.82
Fin de año  6  Depreciación:  727.27  Valor actual:  1454.55
```

```
Fin de año  7  Depreciación:  581.82  Valor actual:   872.73
Fin de año  8  Depreciación:  436.36  Valor actual:   436.36
Fin de año  9  Depreciación:  290.91  Valor actual:   145.45
Fin de año 10  Depreciación:  145.45  Valor actual:     0.00

¿Más cálculos? (S/N) s
¿Introducir un nuevo conjunto de datos? s

Valor original: 5000
Número de años: 4

Método: (1-LR 2-BDD 3-SDA) 1

Método de línea recta

Fin de año  1  Depreciación: 1250.00  Valor actual:  3750.00
Fin de año  2  Depreciación: 1250.00  Valor actual:  2500.00
Fin de año  3  Depreciación: 1250.00  Valor actual:  1250.00
Fin de año  4  Depreciación: 1250.00  Valor actual:     0.00

¿Más cálculos? (S/N) s
¿Introducir un nuevo conjunto de datos? n

Método: (1-LR 2-BDD 3-SDA) 2

Método de balance doblemente declinante

Fin de año  1  Depreciación: 2500.00  Valor actual:  2500.00
Fin de año  2  Depreciación: 1250.00  Valor actual:  1250.00
Fin de año  3  Depreciación:  625.00  Valor actual:   625.00
Fin de año  4  Depreciación:  312.50  Valor actual:   312.50

¿Más cálculos? (S/N) n

Hasta luego y que tenga un buen día
```

Observe que se han procesado dos conjuntos diferentes de datos de entrada. La depreciación para el primer conjunto se calcula utilizando los tres métodos, y para el segundo conjunto utilizando sólo los dos primeros. Por tanto, no es necesario reintroducir los datos de entrada para simplemente recalcular la depreciación utilizando un método diferente.

Los argumentos array se pasan de forma diferente que los elementos con un único valor. Si se especifica el nombre de un array como argumento real, no se hace una copia para la función de los elementos del array. En lugar de esto, lo que se hace es pasar a la función la *posición* del array (la posición del primer elemento). Si se accede a un elemento del array dentro de la función, el acceso hará referencia a la posición de ese elemento del array con relación a la posición del primer elemento. Por tanto, *cualquier alteración de un elemento del array dentro de la función afectará a la rutina que hizo la llamada*. Discutiremos esto con más detalle en el Capítulo 9, cuando tratemos formalmente los arrays.

También se pueden pasar como argumentos a una función otros tipos de estructuras de datos. Discutiremos la transferencia de tales argumentos en capítulos posteriores, cuando se introduzcan estructuras de datos adicionales.

7.6. RECURSIVIDAD

Se llama *recursividad* a un proceso mediante el que una función se llama a sí misma de forma repetida, hasta que se satisface alguna condición determinada. El proceso se utiliza para cálculos repetitivos en los que cada acción se determina en función de un resultado anterior. Se pueden escribir en esta forma muchos problemas iterativos (repetitivos).

Se deben cumplir dos condiciones para que un problema se pueda resolver recursivamente. Primero, el problema se debe escribir en forma recursiva, y segundo, la especificación del problema debe incluir una condición de fin. Supongamos, por ejemplo, que deseamos calcular el factorial de una cantidad entera positiva. Expresaríamos normalmente el problema así: $n! = 1 \times 2 \times 3 \times \times \ldots \times n$, en donde n es el entero positivo especificado (ver Ejemplo 7.5). Sin embargo, podemos también expresar el problema de otra forma, escribiendo $n! = n \times (n - 1)!$ Ésta es la forma recursiva, en la que la acción deseada (el cálculo de $n!$) se expresa en términos de un resultado anterior [el valor de $(n - 1)!$, que se supone conocido]. También sabemos que $1! = 1$ por definición. Esta última expresión proporciona una condición de fin para el proceso resursivo.

EJEMPLO 7.14. Cálculo de factoriales. En el Ejemplo 7.10 vimos dos versiones de un programa que calculaba el factorial de una cantidad dada, utilizando una función no recursiva que se ocupaba de realizar los cálculos en sí. He aquí un programa que realiza el mismo cálculo de forma recursiva.

```
/* calcular el factorial de una cantidad entera utilizando recursi-
   vidad */

#include <stdio.h>

long int factorial (int n);    /* prototipo de función */

main()

{
   int n;

   /* leer la cantidad entera */

   printf("n = ");
   scanf("%d", &n);

   /* calcular y visualizar el factorial */

   printf("n! = %ld\n", factorial(n));
}
```

```
long int factorial(int n)      /* calcular el factorial */

{
   if (n <= 1)
       return(1);
   else
       return(n * factorial(n - 1));

}
```

La parte main del programa simplemente lee la cantidad entera n y llama a continuación a la función factorial. (Recuerde que utilizamos enteros largos para este cálculo porque los factoriales son cantidades enteras muy grandes, aun para valores pequeños de n.) La función factorial se llama a sí misma recursivamente, con un argumento real ($n - 1$) que decrece en cada llamada sucesiva. Las llamadas recursivas terminan cuando el valor del argumento real se hace igual a 1.

Observe que esta forma de la función factorial es más sencilla que la mostrada en el Ejemplo 7.10. La correspondencia entre esta función y la definición original del problema, en términos recursivos, debe resultar inmediata. En particular, observe que la instrucción if - else incluye una condición de terminación que se cumple cuando el valor de n es menor o igual a 1. (Tenga en cuenta que el valor de n nunca será menor que 1, a menos que se introduzca en la computadora un valor inicial inadecuado.)

Cuando se ejecuta el programa, se accede a la función factorial repetidamente, una vez en main y ($n - 1$) veces desde dentro de ella misma, aunque la persona que utilice el programa no se dará cuenta de esto. Sólo se visualizará la respuesta final, por ejemplo,

```
n = 10

n! = 3628800
```

Cuando se ejecuta un programa recursivo, las llamadas recursivas no se ejecutan inmediatamente. Lo que se hace es colocarlas en una *pila* hasta que se encuentra la condición de terminación de la recursividad *. Entonces se ejecutan las llamadas a la función en orden inverso a como se generaron, como si se fueran «sacando» de la pila. Por tanto, cuando se evalúa un factorial recursivamente, las llamadas a las funciones se generarán en el siguiente orden.

$n! = n \times (n-1)!$
$(n-1)! = (n-1) \times (n-2)!$
$(n-2)! = (n-2) \times (n-3)!$
....
$2! = 2 \times 1!$

* Una *pila* es una estructura «last-in, first-out» (último en entrar, primero en salir) en la que los datos sucesivos se «colocan encima» de los anteriores. Los datos se «sacan» después en orden inverso de la pila, como indica la designación «last-in, first-out».

Los valores reales se devolverán en orden inverso, es decir,

$1! = 1$
$2! = 2 \times 1! = 2 \times 1 = 2$
$3! = 3 \times 2! = 3 \times 2 = 6$
$4! = 4 \times 3! = 4 \times 6 = 24$
....
$n! = n \times (n-1)! = \ldots$

El orden inverso de ejecución es una característica típica de todas las funciones recursivas.

Si una función recursiva contiene variables locales, se creará un conjunto *diferente* de variables locales durante cada llamada. Los nombres de las variables locales serán, por supuesto, siempre los mismos, como se hayan declarado en la función. Sin embargo, las variables representarán un conjunto diferente de valores cada vez que se ejecute la función. Cada conjunto de valores se almacenará en la pila; así se podrá disponer de ellas cuando el proceso recursivo se «deshaga», es decir, cuando las llamadas a la función se «saquen» de la pila y se ejecuten.

EJEMPLO 7.15. Escritura inversa. El siguiente programa lee una línea de texto carácter a carácter y escribe los caracteres en orden inverso. El programa utiliza recursividad para la escritura inversa.

```
/*  leer una línea de texto y escribirla en orden inverso utilizando
    recursividad */

#include <stdio.h>

#define EOLN '\n'

void inverso(void);        /* prototipo de función */

main()

{
  printf("Introduce una línea de texto debajo\n");
  inverso();

}

void inverso(void)

/* leer una línea de caracteres y escribirla en orden inverso */

{
  char c;

  if((c = getchar()) != EOLN) inverso();
  putchar(c);
  return;
}
```

La función `main` en este programa simplemente se ocupa de presentar un rótulo y llamar a la función `inverso`, iniciando el proceso recursivo. La función recursiva `inverso` procede entonces a leer caracteres hasta que se encuentre la condición final de línea (`\n`). Cada llamada a esta función hace que se introduzca en la pila un nuevo carácter (un valor nuevo de `c`). Una vez que se encuentra el final de la línea, se sacan los caracteres de la pila y se visualizan en la forma «last-in, first-out» (último en entrar, primero en salir). De esta forma, los caracteres se visualizan de forma inversa a como se introdujeron.

Supongamos que se ejecuta el programa y se introduce la siguiente línea:

```
Tu mirada me recuerda el fulgor de la aurora
```

La salida correspondiente será

```
arorua al ed rogluf le adreucer em adarim uT
```

Algunas veces se puede escribir recursivamente un determinado proceso repetitivo de forma muy concisa, aunque la lógica subyacente puede resultar de difícil comprensión.

EJEMPLO 7.16. Las Torres de Hanoi. Las *Torres de Hanoi* son un conocido juego de niños, que se juega con tres pivotes y un cierto número de discos de diferentes tamaños. Los discos tienen un agujero en el centro, con lo que se pueden apilar en cualquiera de los pivotes. Inicialmente, los discos se amontonan en el pivote de la izquierda por tamaño decreciente, es decir, el más grande abajo y el más pequeño arriba, como se ilustra en la Figura 7.1.

El objeto del juego es conseguir llevar los discos del pivote más a la izquierda al de más a la derecha, sin colocar nunca un disco sobre otro más pequeño. Sólo se puede mover un disco cada vez, y cada disco debe encontrarse siempre en uno de los pivotes.

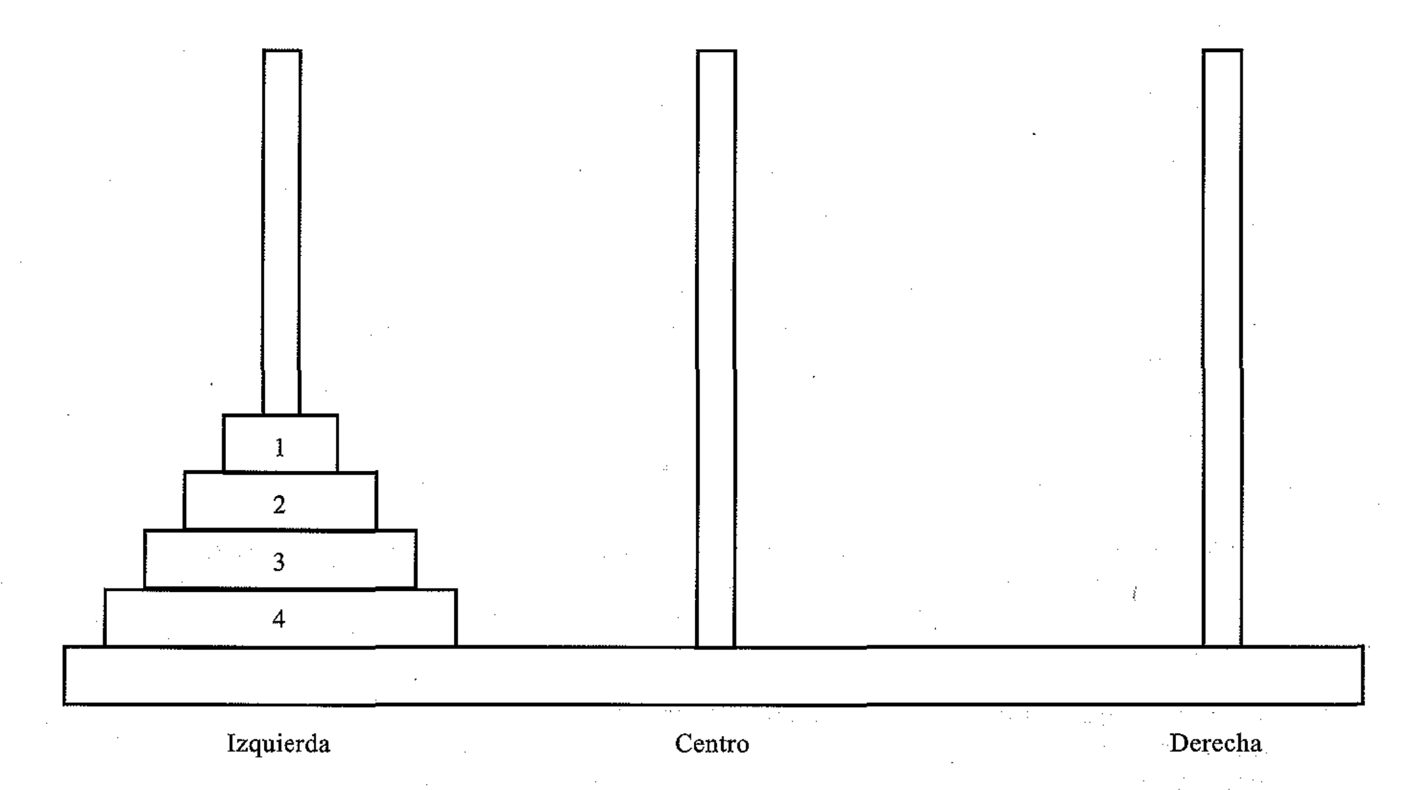


Figura 7.1.

La estrategia general a seguir es considerar uno de los pivotes como origen y el otro como destino. El tercer pivote se utilizará para almacenamiento auxiliar, con lo que podremos mover los discos sin poner ninguno sobre otro más pequeño. Supongamos que hay *n* discos, numerados del más pequeño al más grande, como en la Figura 7.1. Si los discos se encuentran apilados inicialmente en el pivote izquierdo, el problema de mover los *n* discos al pivote derecho se puede formular de la siguiente forma recursiva:

1. Mover los *n*–1 discos superiores del pivote izquierdo al del centro.
2. Mover el *n*-ésimo disco (el más grande) al pivote de la derecha.
3. Mover los *n*–1 discos del pivote del centro al de la derecha.

El problema se puede resolver de esta manera para cualquier valor de *n* mayor que 0 (*n*=0 representa la condición de fin).

Con vistas a la redacción del programa, primero etiquetaremos los pivotes, de forma que el pivote izquierdo se representará por `I`, el del centro por `C` y el derecho por `D`. Construiremos entonces una función recursiva llamada `transferir` que transferirá *n* discos de un pivote a otro. Refirámonos a los pivotes origen, destino y de almacenamiento temporal con las variables `desde`, `hacia` y `temp`, respectivamente. Como resultado de esto, si asignamos el carácter `I` a `desde`, `D` a `hacia` y `C` a `temp`, estaremos especificando el movimiento de *n* discos del pivote izquierdo al derecho, utilizando el pivote del centro para almacenamiento intermedio.

Con esta notación, la estructura de la función será:

```
void transferir(int n, char desde, char hacia, char temp)
/*     n = número de discos
   desde = origen
   hacia = destino
    temp = almacenamiento temporal */

{
  if (n > 0) {

     /* mover n-1 discos desde su origen hasta el pivote temporal */

     /* mover el n-ésimo disco desde su origen hasta su destino */

     /* mover n-1 discos desde el pivote temporal hasta su destino */
  }
}
```

La transferencia de los *n*–1 discos se puede realizar mediante una llamada recursiva a `transferir`. Por tanto, podemos escribir

```
transferir(n-1, desde, temp, hacia);
```

para la primera transferencia, y

```
transferir(n-1, temp, hacia, desde);
```

para la segunda. (*Observe el orden de los argumentos en cada llamada.*) El movimiento del disco *n*-ésimo del origen al destino simplemente requiere escribir los valores en curso de `desde` y `hacia`. La función completa se puede escribir como sigue:

```
void transferir(int n, char desde, char hacia, char temp)

/* transferir n discos de un pivote a otro */

/*      n = número de discos
    desde = origen
    hacia = destino
     temp = almacenamiento temporal */

{
  if (n > 0) {
     /* mover n-1 discos desde su origen hasta el pivote temporal */
     transferir(n-1, desde, temp, hacia);

     /* mover el n-ésimo disco desde su origen hasta su destino */
     printf("Mover disco %d desde %c hasta %c\n", n, desde, hacia);

     /* mover n-1 discos desde el pivote temporal hasta su destino */
     transferir(n-1, temp, hacia, desde);
  }
  return;
}
```

El añadir la parte `main` del programa es ya una cuestión sencilla, simplemente se debe leer el valor de `n` e iniciar los cálculos llamando a `transferir`. En esta primera llamada a la función los parámetros reales se especifican como constantes de carácter, esto es,

```
transferir(n, 'I', 'D', 'C');
```

Esta llamada a función especifica la transferencia de `n` discos del pivote de la izquierda (el origen) al de la derecha (el destino), utilizando el pivote del centro para almacenamiento auxiliar.

He aquí el programa completo.

```
/* las TORRES DE HANOI - resolución utilizando recursividad */

#include <stdio.h>

void transferir(int n, char desde, char hacia, char temp);
                                        /* prototipo de función */

main()
{
  int n;

  printf("Bienvenido a las TORRES DE HANOI\n\n");
  printf("¿Cuantos discos? ");
  scanf("%d", &n);
  printf("\n");
  transferir(n,'I','D','C');
}

void transferir(int n, char desde, char hacia, char temp)
```

```
/* transferir n discos de un pivote a otro */

/* n     = número de discos
   desde = origen
   hacia = destino
   temp  = almacenamiento temporal */
{
   if (n > 0) {
      /* mover n-1 discos desde su origen hasta el pivote temporal */
      transferir(n-1, desde, temp, hacia);

      /* mover el n-ésimo disco desde su origen hasta su destino */
      printf("Mover disco %d desde %c hasta %c\n", n, desde, hacia);

      /* mover n-1 discos desde el pivote temporal hasta su destino */
      transferir(n-1, temp, hacia, desde);

   }
   return;

}
```

Debe quedar claro que la función `transferir` recibe un conjunto diferente de valores de sus argumentos cada vez que se llama a la función. Este conjunto de valores se almacenará en la pila de forma que cada conjunto es independiente de los otros. Durante la ejecución del programa se sacan a su debido tiempo de la pila. Es la capacidad de almacenar y recuperar estos conjuntos independientes de valores lo que permite que funcione el proceso recursivo.

Cuando se ejecuta el programa para el caso en que n=3, se obtiene la siguiente salida:

```
Bienvenido a las TORRES DE HANOI

¿Cuantos discos? 3

Mover disco 1 desde I hasta D
Mover disco 2 desde I hasta C
Mover disco 1 desde D hasta C
Mover disco 3 desde I hasta D
Mover disco 1 desde C hasta I
Mover disco 2 desde C hasta D
Mover disco 1 desde I hasta D
```

Estudiar detenidamente estos movimientos para comprobar que la solución es correcta. La lógica del programa es complicada a pesar de su aparente sencillez.

Veremos otro ejemplo de programación que utiliza la recursividad en el Capítulo 11, cuando tratemos las listas enlazadas.

El uso de la recursividad no es necesariamente la mejor aproximación a un problema, aunque la definición del problema sea recursiva. Una implementación no recursiva puede ser más eficiente, en cuanto a utilización de memoria y velocidad de ejecución. Como conclusión, el uso de la recursividad puede involucrar un compromiso entre simplicidad y rendimiento. Cada problema se debe juzgar individualmente teniendo en cuenta sus características específicas.

CUESTIONES DE REPASO

7.1. ¿Qué es una función? ¿Se requiere la utilización de funciones al escribir un programa en C?

7.2. Citar tres ventajas de la utilización de funciones.

7.3. ¿Qué se entiende por una llamada a una función? ¿Desde qué partes de un programa se puede llamar a una función?

7.4. ¿Qué son los argumentos? ¿Cuál es su propósito? ¿Qué otro término se utiliza a veces en lugar de argumento?

7.5. ¿Cuál es la finalidad de la instrucción `return`?

7.6. ¿Cuáles son las dos principales componentes de una definición de función?

7.7. ¿Cómo se escribe la primera línea de una definición de función? ¿Cuál es el propósito de cada elemento o cada grupo de elementos?

7.8. ¿Qué son los argumentos formales? ¿Qué son los argumentos reales? ¿Cuál es la relación entre ambos tipos de argumentos?

7.9. Citar otros términos que se utilizan en lugar de *argumento formal* y *argumento real.*

7.10. ¿Pueden coincidir los nombres de los argumentos formales dentro de una función con los nombres de otras variables definidas fuera de la función? Explicarlo.

7.11. ¿Pueden coincidir los nombres de los argumentos formales dentro de una función con los nombres de otras variables definidas dentro de la función? Explicarlo y comparar la respuesta con la última pregunta.

7.12. Citar las reglas relacionadas con el uso de la instrucción `return`. ¿Se pueden incluir varias expresiones en una instrucción `return`? ¿Se pueden incluir varias instrucciones `return` en una función?

7.13. ¿Qué relación debe existir entre el tipo de datos que aparece al comienzo de la primera línea de la definición de una función y el valor devuelto por la instrucción `return`?

7.14. ¿Por qué se puede incluir una instrucción `return` en una función que no devuelve ningún valor?

7.15. ¿Cuál es la finalidad de la palabra reservada `void`? ¿Dónde se utiliza esta palabra reservada?

7.16. Citar las reglas relacionadas con la llamada a funciones. ¿Qué relación debe existir entre los argumentos reales y los formales correspondientes en la definición de la función? ¿Están sujetos a las mismas restricciones los argumentos reales que los formales?

7.17. ¿Se puede llamar a una función desde más de un lugar en un programa?

7.18. ¿Qué son los prototipos de funciones? ¿Cuál es su propósito? ¿Dónde se colocan normalmente los prototipos de funciones en un programa?

7.19. Citar las reglas asociadas con los prototipos de funciones. ¿Cuál es la finalidad de cada elemento o grupo de elementos?

7.20. ¿Cómo se especifican los tipos de datos de los argumentos en un prototipo de función? ¿Qué sentido tiene el incluir los tipos de datos de los argumentos en un prototipo de función?

7.21. Cuando se accede a una función, ¿deben coincidir los nombres de los argumentos reales con los nombres de los argumentos en el correspondiente prototipo de función?

7.22. Suponer que la función `F1` llama a la función `F2` dentro de un programa en C. ¿Tiene importancia el orden en el que se definan las funciones? Explicarlo.

7.23. Describir la forma en que los argumentos reales pasan información a una función. ¿Qué nombre se asocia a este proceso? ¿Cuáles son las ventajas e inconvenientes de pasar los argumentos de esta manera?

7.24. ¿Qué diferencias hay entre pasar un array a una función y pasar un dato simple a una función?

7.25. Suponer que se pasa un array a una función como argumento. Si dentro de la función se altera el valor de un elemento del array, ¿se reconocerá este cambio en la parte del programa que llamó a la función?

7.26. ¿Qué es la recursividad? ¿Qué ventajas tiene hacer uso de ella?

7.27. Explicar por qué algunos programas pueden ser resueltos con o sin recursividad.

7.28. ¿Qué es una pila? ¿En qué orden se añade y elimina información de la pila?

7.29. Explicar lo que sucede cuando se ejecuta un programa que contiene llamadas recursivas, en términos de la información añadida o eliminada de la pila.

7.30. Cuando se ejecuta un programa que contiene llamadas recursivas, ¿cómo se interpretan las variables locales dentro de una función recursiva?

7.31. Si se programa recursivamente un proceso iterativo, ¿será necesariamente más eficiente el programa que el correspondiente a una versión no recursiva?

PROBLEMAS

7.32. Explicar el significado de cada uno de los siguientes prototipos de funciones.

a) `int f(int a);`
b) `double f(double a, int b);`
c) `void f(long a, short b, unsigned c);`
d) `char f(void);`
e) `unsigned f(unsigned a, unsigned b);`

7.33. Cada una de las siguientes líneas es la primera línea de una definición de función. Explicar el significado de cada una.

a) `float f(float a, float b)`
b) `long f(long a)`
c) `void f(int a)`
d) `char f(void)`

7.34. Escribir una llamada a función (acceso a función) apropiada para cada una de las siguientes funciones.

a)
```
float formula(float x)
{
  float y;

  y = 3 * x - 1;
  return(y);
}
```

b)
```
void escribe(int a, int b)
{
   int c;

   c = sqrt(a * a + b * b);
   printf("c = %i\n", c);
}
```

7.35. Escribir la primera línea de la definición de la función, incluyendo las declaraciones de los argumentos formales, para cada una de las situaciones que se describen a continuación.

a) Una función llamada `muestra` genera y devuelve una cantidad entera.
b) Una función llamada `raiz` acepta dos argumentos enteros y devuelve un resultado en coma flotante.
c) Una función llamada `convertir` acepta un carácter y devuelve un carácter.
d) Una función llamada `transferir` acepta un entero largo y devuelve un carácter.
e) Una función llamada `inversa` acepta un carácter y devuelve un entero largo.
f) Una función llamada `procesar` acepta un entero y dos cantidades en coma flotante (en este orden) y devuelve una cantidad en doble precisión.
g) Una función llamada `valor` acepta dos cantidades en doble precisión y un entero corto (en este orden). Las cantidades de entrada se procesan para generar un valor de doble precisión que se escribe como resultado final.

7.36. Escribir prototipos de funciones adecuados para cada uno de los esquemas que se muestran a continuación.

a)
```
main()
{
   int a, b, c;
   . . .
   c = func1(a, b);
   . . .
}

int func1(int x, int y)
{
   int z;

   z = . . .;
   return(z);
}
```

b)
```
main()
{
   double a, b, c;
   . . .
   c = func1(a, b);
   . . .
}
```

```
double func1(double x, double y)
{
  double z;

  z = . . .;
  return(z);
}
```

c)
```
main()
{
  int a;
  float b;
  long int c;
  . . .
  c = func1(a, b);
  . . .
}

long int func1(int x, float y)
{
   long int z;

   z = . . .;
   return(z);
 }
```

d)
```
main()
{
  double a, b, c, d;
  . . .
  c = func1(a, b);
  . . .
  d = func2(a + b, a + c);
}

double func1(double x, double y)
{
  double z;

  z = 10 * func2(x, y);
  return(z);
}

double func2(double x, double y)
{
  double z;

  z = . . .;
  return(z);
}
```

7.37. Describir la salida generada por cada uno de los siguientes programas:

a)
```
#include <stdio.h>

int func1(int cont);

main()
{
    int a, cont;

    for (cont = 1; cont <= 5; ++cont) {
        a = func1(cont);
        printf("%d  ", a);
    }
}

int func1(int x);
{
    int y;

    y = x * x;
    return(y);
}
```

b) Escribir el programa anterior de alguna forma más concisa.

c)
```
#include <stdio.h>

int func1(int n);

main()
{
    int n = 10;

    printf("%d", func1(n));
}

int func1(int n);
{
    if (n > 0) return(n + func1(n - 1));
}
```

d)
```
#include <stdio.h>

int func1(int n);

main()
{
    int n = 10;

    printf("%d", func1(n));
}
```

```
int func1(int n);
{
  if (n > 0) return(n + func1(n - 2));
}
```

7.38. Expresar cada una de las siguientes fórmulas de forma recursiva:

a) $y = (x_1 + x_2 + ... + x_n)$
b) $y = 1 - x + x^2/2 - x^3/6 + x^4/24 + ... + (-1)^n x^n/n!$
c) $p = (f_1 * f_2 * ... * f_t)$

PROBLEMAS DE PROGRAMACIÓN

7.39. Escribir una función que calcule y presente las raíces reales de la ecuación de segundo grado

$ax^2 + bx + c = 0$

utilizando la fórmula

$$x = \frac{-b \pm (b^2 - 4ac)^{1/2}}{2a}$$

Suponer que *a*, *b* y *c* son argumentos en coma flotante con valores dados y que x_1 y que x_2 son variables en coma flotante. Suponer también que $b^2 > 4*a*c$, de forma que las raíces calculadas son siempre reales.

7.40. Escribir un programa completo en C que calcule las raíces de la ecuación de segundo grado

$ax^2 + bx + c = 0$

utilizando la fórmula cuadrática que se describió en el problema anterior. Leer los coeficientes *a*, *b* y *c* en la función `main`. Acceder a continuación a la función escrita para el problema anterior para conseguir la solución deseada. Finalmente, escribir los valores de los coeficientes, seguidos de los valores calculados de x_1 y x_2. Asegurarse de que la salida aparece con rótulos claros.
Comprobar el programa con los siguientes datos:

a	*b*	*c*
2	6	1
3	3	0
1	3	1

7.41. Modificar la función escrita para el Problema 7.39 de forma que se calculen *todas* las raíces de la ecuación de segundo grado

$ax^2 + bx + c = 0$

habiendo dado los valores de a, b y c. Observar que las raíces estarán repetidas (sólo habrá una raíz real) si $b^2 = 4*a*c$. Además, las raíces serán complejas si $b^2 < 4*a*c$. En este caso la parte real de cada raíz es igual a

$-b/(2*a)$

y las partes imaginarias se calculan con

$\pm[(4ac - b^2)^{1/2}]i$

en donde i representa $(-1)^{1/2}$.

7.42. Modificar el programa en C escrito para el Problema 7.40 de forma que se calculen *todas* las raíces de la ecuación de segundo grado

$ax^2 + bx + c = 0$

utilizando la función escrita en el Problema 7.41. Asegurarse de que toda la salida se acompaña de rótulos claros. Comprobar el programa utilizando los siguientes datos:

a	b	c
2	6	1
3	3	0
1	3	1
0	12	–3
3	6	3
2	–4	3

7.43. Escribir una función que permita elevar un número en coma flotante a una potencia entera. En otras palabras, deseamos evaluar la fórmula

$y = x^n$

en donde y y x son variables en coma flotante y n una variable entera.

7.44. Escribir un programa completo en C que lea los valores de x y n, evalúe la fórmula

$y = x^n$

utilizando la función escrita en el Problema 7.43 y escriba a continuación el resultado. Comprobar el programa utilizando los siguientes datos:

x	n	x	n
2	3	1.5	3
2	12	1.5	10
2	–5	1.5	–5
–3	3	0.2	3
–3	7	0.2	5
–3	–5	0.2	–5

7.45. Ampliar la función escrita para el Problema 7.43 para que se puedan elevar valores positivos de x a *cualquier* potencia, entera o en coma flotante. (*Sugerencia:* utilizar la fórmula

$y = x^n = e^{(n \ln x)}$

Recordar incluir una comprobación de que el valor de x sea adecuado.)

Incluir esta función en el programa escrito para el Problema 7.44. Comprobar el programa utilizando los datos dados en el Problema 7.44 y los siguientes datos adicionales:

x	n	x	n
2	0.2	1.5	0.2
2	−0.8	1.5	−0.8
−3	0.2	0.2	0.2
−3	−0.8	0.2	−0.8
		0.2	0.0

7.46. Modificar el programa para calcular la solución de una ecuación algebraica, dado en el Ejemplo 6.22, de forma que cada iteración se realice en una función. Compilar y ejecutar el programa para asegurarse de que funciona correctamente.

7.47. Modificar el programa del Ejemplo 6.17 para hacer la media de una lista de números, de forma que haga uso de funciones para leer números y devolver su suma. Comprobar el programa utilizando los siguientes diez números:

27.5	87.0
13.4	39.9
53.8	47.7
29.2	8.1
74.5	63.2

7.48. Modificar el programa del Ejemplo 5.2 para calcular el interés compuesto de forma que los cálculos se realicen en una función definida por el programador. Escribir la función de forma tal que los valores de P, r y n se pasen como argumentos y se devuelva el valor calculado de F. Comprobar el programa utilizando los siguientes datos:

P	r	n
1000	6	20
1000	6.25	20
333.33	8.75	20
333.33	8.75	22.5

7.49. Para cada uno de los siguientes problemas, escribir un programa completo en C que incluya una función recursiva.

a) Los *polinomios de Legendre* se pueden calcular mediante las fórmulas $P_0 = 1$, $P_1 = x$,

$$P_n = [(2n - 1)/n]\, x\, P_{n-1} - [(n - 1)/n]\, P_{n-2}$$

en donde $n = 2, 3, 4, \ldots$, y x es un número en coma flotante entre −1 y 1. (Advertir que los coeficientes de los polinomios de Legendre son cantidades en coma flotante.)
Generar los n primeros polinomios. Los valores de n y x deben ser parámetros de entrada.

b) Determinar la suma de n números en coma flotante [ver Problema 7.38(*a*)]. Leer un nuevo número en cada llamada a una función recursiva.

c) Evaluar los n primeros términos de las series especificadas en el Problema 7.38(*b*). Introducir n como parámetro de entrada.

d) Determinar el producto de n números en coma flotante [ver Problema 7.38(*c*)]. Leer un nuevo número durante cada llamada a una función recursiva.

Al final del Capítulo 8 se pueden encontrar otros problemas de programación referentes al uso de funciones.

CAPÍTULO 8

Estructura de un programa

Este capítulo considera varios temas asociados con la estructura de programas que constan de más de una función. Trataremos en primer lugar la diferencia entre variables «locales» que son reconocidas solamente dentro de una función, y variables «globales» que son reconocidas en dos o más funciones. En este capítulo se verá cómo definir y utilizar variables globales.

Además, se considera el problema de la retención de información estática o dinámica mediante una variable local. Una variable local normalmente no conserva su valor una vez que el control del programa se ha transferido fuera de su función de definición. Sin embargo, a veces interesa que algunas variables retengan sus valores de modo que se pueda volver más tarde a la función y reanudar el cómputo.

Finalmente, puede resultar interesante desarrollar un programa multifunción grande en términos de varios archivos independientes, con un número pequeño de funciones (quizás una sola) definidas dentro de cada archivo. En tales programas las funciones individuales pueden ser definidas y accedidas localmente dentro de un archivo, o globalmente dentro de múltiples archivos. Esto es similar a la definición y el uso de variables locales versus globales en un programa multifunción de un único archivo.

8.1. TIPOS DE ALMACENAMIENTO

Ya se ha mencionado que hay dos formas diferentes de caracterizar variables: por su *tipo de datos* y por *tipo de almacenamiento* (ver sección 2.6). El tipo de datos se refiere al tipo de información representada por una variable, por ejemplo, un número entero, un número en coma flotante, un carácter, etc. El tipo de almacenamiento se refiere a la permanencia de la variable y a su *ámbito* dentro del programa, que es la parte del programa en la que se reconoce la variable.

Hay cuatro especificaciones diferentes de tipo de almacenamiento en C: *automática*, *externa*, *estática* y *registro*. Están identificadas por las siguientes palabras reservadas `auto`, `extern`, `static` y `register`, respectivamente. En este capítulo se tratarán los tipos de almacenamiento *automático*, *externo* y *estático*. El tipo de almacenamiento *registro* se discute en la sección 13.1.

A veces se puede establecer el tipo de almacenamiento asociado a una variable simplemente por la posición de su declaración en el programa. En otras situaciones, sin embargo, la palabra reservada que especifica un tipo particular de almacenamiento se tiene que colocar al comienzo de la declaración de la variable.

EJEMPLO 8.1. A continuación se muestran varias declaraciones típicas de variables que incluyen la especificación de un tipo de almacenamiento.

```
auto int a, b, c;

extern float raiz1, raiz2;

static int cont = 0;

extern char asterisco;
```

La primera declaración establece que `a`, `b` y `c` son variables enteras automáticas, y la segunda que `raiz1` y `raiz2` son variables en coma flotante externas. La tercera declaración establece que `cont` es una variable entera estática de valor inicial 0, y en la última declaración se establece `asterisco` como una variable externa de tipo carácter.

El procedimiento exacto para establecer el tipo de almacenamiento para una variable depende del tipo particular de almacenamiento y de la manera en que está organizado el programa (archivo simple frente a múltiples archivos). Consideraremos estas reglas en las siguientes secciones del capítulo.

8.2. VARIABLES AUTOMÁTICAS

Las *variables automáticas* se declaran siempre dentro de la función y son locales a la función donde han sido declaradas; es decir, su ámbito está confinado a esa función. Las variables automáticas definidas en funciones diferentes serán independientes unas de otras, incluso si tienen el mismo nombre.

Cualquier variable declarada dentro de una función se interpreta como una variable automática a menos que se incluya dentro de la declaración un tipo distinto de almacenamiento. Esto incluye la declaración de argumentos formales. Todas las variables encontradas en los ejemplos de los capítulos anteriores han sido variables automáticas.

Como la localización de la declaración de la variable dentro del programa determina el tipo de almacenamiento automático, no se necesita la palabra reservada `auto` al principio de cada declaración de variable. No hay problema para incluir la especificación `auto` dentro de una declaración si el programador lo desea, pero normalmente no se hace.

EJEMPLO 8.2. Cálculo de factoriales. Consideremos de nuevo el programa para calcular factoriales, mostrado originalmente en el Ejemplo 7.10. Dentro de `main`, `n` es una variable automática. Dentro de `factorial`, `i` y `prod`, así como el argumento formal `n`, son variables automáticas.

La designación del tipo de almacenamiento `auto` podría haberse incluido explícitamente en las declaraciones de las variables si se hubiera deseado. Así el programa podría haberse escrito como sigue:

```
/* calcular el factorial de una cantidad entera */

#include <stdio.h>

long int factorial(int n);
```

```
main()

{
   auto int n;

   /* leer la cantidad entera */

   printf("\nn = ");
   scanf("%d", &n);

   /* calcular y visualizar el factorial */

   printf("\nn! = %ld", factorial(n));
}

long int factorial(auto int n)         /* calcular el factorial */

{
   auto int i;
   auto long int prod = 1;

   if (n > 1)
      for (i = 2; i <= n; ++i)
          prod *= i;
   return(prod);
}
```

Cualquiera de los métodos es aceptable. Sin embargo, como regla, la designación `auto` no se incluye en las declaraciones de variables o argumentos formales, ya que es el tipo de almacenamiento por omisión. De este modo, el programa del Ejemplo 7.10 presenta un estilo de programación más usual.

Se pueden asignar valores iniciales a las variables automáticas incluyendo las expresiones adecuadas dentro de la declaración de variables, como en el ejemplo anterior, o por asignación explícita de expresiones en cualquier parte de la función. Tales valores se reasignarán cada vez que se entre en la función. Si una variable automática no es inicializada de alguna manera, su valor inicial será impredecible y probablemente incomprensible.

Una variable automática no mantiene su valor cuando se transfiere el control fuera de la función en que está definida. Por tanto, cualquier valor asignado a una variable automática dentro de una función se perderá una vez que se sale de la función. Si la lógica del programa requiere que un valor particular sea asignado a una variable automática cada vez que se ejecuta la función, ese valor tendrá que reasignarse cada vez que se entre en la función (siempre que se acceda a la función).

EJEMPLO 8.3. Longitud media de varias líneas de texto. Escribamos a continuación un programa en C que lea varias líneas de texto y determine el número medio de caracteres (incluyendo puntuación y espacios en blanco) en cada línea. Estructuraremos el programa de tal manera que continúe leyendo líneas hasta encontrar una línea vacía (una línea cuyo primer carácter sea `\n`).

Utilizaremos una función (`contlinea`) que lee una línea de texto y cuenta el número de caracteres, excluyendo el carácter de nueva línea (`\n`) que marca el fin de la línea. La rutina desde la que se llama

(main) mantiene una suma acumulativa, así como el número total de líneas que se han leído. La función se llamará repetitivamente (leyendo una nueva línea cada vez) hasta que se encuentre una línea vacía. El programa divide entonces el número acumulado de caracteres por el número total de líneas para obtener la media.

He aquí el programa completo.

```
/*  leer varias líneas de texto y determinar el número medio de carac-
    teres por línea */

#include <stdio.h>

int contlinea(void);

main()

{
   int n;             /* número de caracteres en una línea */
   int cont = 0;      /* número de líneas */
   int suma = 0;      /* número total de caracteres */
   float media;       /* número medio de caracteres por línea */

   printf("Introducir el texto debajo:\n");

   /* leer una línea de texto y actualizar los contadores */

   while ((n = contlinea()) > 0)  {
         suma += n;
         ++cont;
   }

   media = (float) suma / cont;
   printf("\nNúmero medio de caracteres por línea: %5.2f", media);
}

int contlinea(void)

/* leer una línea de texto y contar el número de caracteres */

{
   char linea[80];
   int cont = 0;

   while ((linea[cont] = getchar()) != '\n')
         ++cont;
   return (cont);
}
```

Vemos que main contiene cuatro variables automáticas: n, cont, suma y media, mientras que contlinea contiene dos: linea y cont. (Linea es un array de caracteres de 80 elementos que representa el contenido de una línea de texto.) Tres de estas variables tienen asignado el valor inicial cero.

Observe también que `cont` tiene diferente significado dentro de cada función. Dentro de `contlinea`, `cont` representa el número de caracteres en una línea, mientras que en `main`, `cont` representa el número total de líneas que se han leído. Además, `cont` se pone a cero cada vez que se accede a `contlinea`. Esto no afecta al valor de `cont` dentro de `main`, ya que una variable es independiente de la otra. Esto estaría claro si las variables tuvieran nombres distintos, por ejemplo `cont` y `lineas`, o quizás `caracteres` y `lineas`. Se ha usado el mismo nombre para las dos variables para ilustrar la independencia de las variables automáticas en funciones diferentes.

A continuación se muestra una sesión interactiva de la ejecución del programa. Como siempre, las respuestas del usuario están subrayadas.

```
Introducir el texto debajo:
En un lugar de la Mancha de cuyo
nombre no quiero acordarme.

Número medio de caracteres por línea: 29.50
```

Si se desea, el ámbito de una variable automática puede ser menor que la función. De hecho, pueden declararse variables automáticas dentro de una instrucción compuesta. Con programas pequeños no suelen existir ventajas al hacer esto, pero puede ser aconsejable en programas mayores.

8.3. VARIABLES EXTERNAS (GLOBALES)

Las *variables externas*, en contraste con las variables automáticas, no están confinadas a funciones simples. Su ámbito se extiende desde el punto de definición hasta el resto del programa. Por tanto, generalmente abarcan dos o más funciones y frecuentemente todo el programa. A menudo se les llama *variables globales*.

Como las variables externas se reconocen globalmente, se puede acceder a las mismas desde cualquier función que se encuentre dentro de su ámbito. Mantienen los valores asignados dentro de este ámbito. Por tanto, a una variable externa se le puede asignar un valor dentro de una función, y este valor puede usarse (al acceder a la variable externa) dentro de otra función.

La utilización de variables externas proporciona un mecanismo adecuado de transferencia de información entre funciones. En particular, podemos transferir información a una función sin usar argumentos. Esto es especialmente conveniente cuando una función requiere numerosos datos de entrada. Es más, ahora tenemos un método para transferir múltiples datos fuera de una función, porque la instrucción `return` sólo puede transferir un dato. (Veremos otro modo de transferir información entre funciones en el Capítulo 10, donde se tratan los punteros.)

Al trabajar con variables externas hay que distinguir entre *definiciones* de variables externas y *declaraciones* de variables externas. Una *definición* de variable externa se escribe de la misma manera que una declaración de una variable ordinaria. Tiene que aparecer fuera, y normalmente antes, de las funciones que acceden a las variables externas. Una definición de variable externa reserva automáticamente el espacio de almacenamiento requerido en la memoria de la computadora. La asignación de valores iniciales puede incluirse, si se desea, dentro de una definición de variable externa (veremos más detalles sobre esto posteriormente).

No se requiere el especificador de tipo de almacenamiento `extern` en una definición de una variable externa, porque las variables externas se identifican por la localización de su definición en el programa. De hecho, muchos compiladores de C prohíben la aparición del especificador de tipo de almacenamiento `extern` dentro de una definición de variable externa. En este libro seguiremos este convenio.

Si una función requiere una variable que ha sido definida antes en el programa, entonces la función puede acceder libremente a la variable externa, sin ninguna declaración especial dentro de la función. (Recuerde, sin embargo, que *cualquier alteración del valor de la variable externa dentro de la función será reconocido dentro del ámbito completo de la variable externa*.) Por otra parte, si la definición de función *precede* a la definición de variable externa, entonces la función tiene que incluir una *declaración* para la variable externa. Las definiciones de funciones en un programa grande frecuentemente incluyen declaraciones de variables externas; si son o no necesarias es cuestión de una buena práctica de programación.

Una *declaración* de variable externa tiene que comenzar con el especificador de tipo de almacenamiento `extern`. El nombre de la variable externa y su tipo de datos tienen que coincidir con su correspondiente definición de variable externa que aparece fuera de la función. No se reservará espacio de almacenamiento para variables externas como consecuencia de una declaración de variable externa. Es más, una declaración de variable externa *no puede* incluir una asignación de valores iniciales. Éstas son las diferencias importantes entre la *definición* de una variable externa y la *declaración* de una variable externa.

EJEMPLO 8.4. Búsqueda de un máximo. Supongamos que queremos encontrar el valor particular de x que hace máxima la función

```
y = x cos(x)
```

en el intervalo limitado por $x = 0$ a la izquierda y $x = \pi$ a la derecha. Necesitaremos conocer con bastante exactitud el valor maximizante de x. Necesitamos también que el esquema de búsqueda sea relativamente eficaz en el sentido que la función $y = x \cos(x)$ se debe evaluar el menor número de veces posible.

Una forma obvia de resolver este problema sería generar un gran número de funciones de prueba muy próximas (esto es, evaluar $x = 0$, $x = 0.0001$, $x = 0.0002$, ..., $x = 3.1415$ y $x = 3.1416$) y determinar el mayor de éstos por inspección visual. Esto no sería muy eficiente, sin embargo, y requeriría de intervención humana para obtener el resultado final. En su lugar, utilizaremos el siguiente *esquema de eliminación*, que es un procedimiento computacional altamente eficiente para todas las funciones que sólo tienen un máximo (un solo «pico») dentro del intervalo de búsqueda.

El cálculo se realizará como sigue. Empezamos con dos puntos de búsqueda en el centro del intervalo de búsqueda, marcando una muy pequeña distancia entre ellos, como se muestra en la Figura 8.1.

Se emplea la siguiente notación:

`a` = extremo izquierdo del intervalo
`xi` = punto interior de búsqueda a la izquierda
`xd` = punto interior de búsqueda a la derecha
`b` = extremo derecho del intervalo
`sep` = distancia entre `xi` y `xd`.

Si se conocen `a`, `b` y `sep`, entonces los puntos interiores pueden calcularse como

```
xi = a + .5 * (b - a - sep)

xd = a + .5 * (b - a + sep) = xi + sep
```

Evaluemos la función `y = x cos(x)` en `xi` y `xd`. Llamemos a estos valores `yi` e `yd`, respectivamente. Supongamos que `yi` resulta ser mayor que `yd`. Entonces el máximo está en alguna parte entre `a` y `xd`. Por tanto, se retendrá sólo esta parte del intervalo de búsqueda que va desde `x = a` hasta `x = xd`. Ahora nos referiremos al punto viejo `xd` como `b`, pues éste es el extremo derecho del nuevo intervalo de búsqueda, y se generan otros dos *nuevos* puntos de búsqueda `xi` y `xd`. Estos puntos se localizarán en el centro del nuevo intervalo de búsqueda, separados una distancia `sep`, como se muestra en la Figura 8.2.

Por otro lado, supongamos ahora que en nuestro intervalo *original* de búsqueda el valor de `yd` resulta mayor que `yi`. Esto indicaría que nuestro intervalo de búsqueda se halla entre `xi` y `b`. Por tanto, renombramos el punto originalmente llamado `xi` como `a` y generamos dos nuevos puntos de búsqueda, `xi` y `xd`, en el centro del nuevo intervalo de búsqueda, como se muestra en la Figura 8.3.

Continuamos generando un nuevo par de puntos de búsqueda en el centro de cada nuevo intervalo, comparando los respectivos valores de `y`, y eliminando una parte del intervalo de búsqueda hasta que el intervalo de búsqueda se hace menor que `3 * sep`. Llegado este punto, no se pueden distinguir los puntos interiores de los límites. Por tanto finaliza la búsqueda.

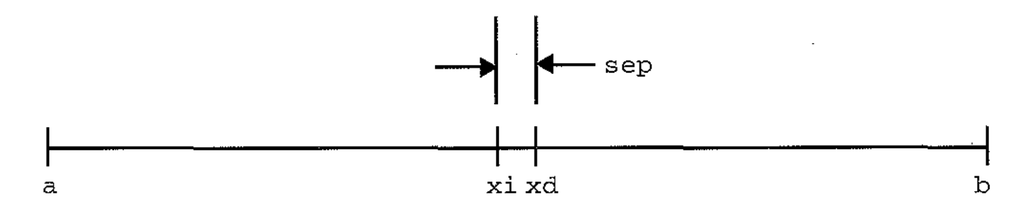


Figura 8.1.

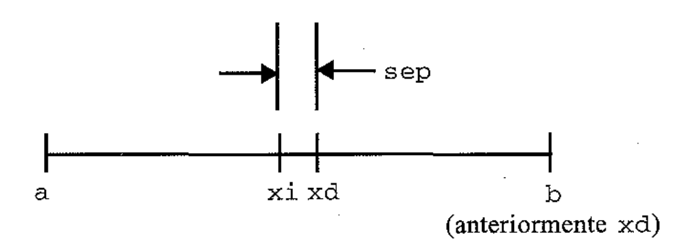


Figura 8.2.

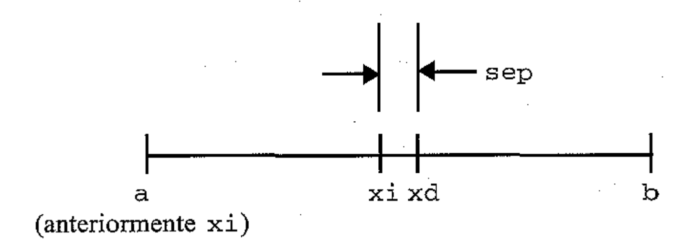


Figura 8.3.

Cada vez que hacemos una comparación entre `yi` e `yd`, eliminamos la parte del intervalo de búsqueda que contiene el valor más pequeño de `y`. Si ambos valores interiores son idénticos (lo cual puede suceder pero es inusual), entonces el procedimiento de búsqueda se detiene y se supone que el máximo tiene lugar en el centro de los dos últimos puntos internos.

Una vez acabada la búsqueda, tanto porque el intervalo de búsqueda se ha hecho lo suficientemente pequeño o porque los dos puntos interiores tienen valores idénticos de `y`, se puede calcular la localización aproximada del máximo como

```
xmax = 0.5 * (xi + xd)
```

El valor máximo correspondiente de la función se puede obtener como `xmax * cos(xmax)`.

Consideremos un esquema de programa para el caso general en que `a` y `b` son cantidades de entrada pero `sep` tiene un valor fijo de `0.0001`.

1. Asignar un valor `sep = 0.0001`.
2. Leer los valores de `a` y `b`.
3. Repetir lo siguiente hasta que `yi` se haga igual a `yd` (el máximo estará en el punto medio), o el valor más reciente de `(b - a)` sea menor o igual que `(3 * sep)`:
   a) Generar los dos puntos interiores, `xi` y `xd`.
   b) Calcular los correspondientes valores de `yi` e `yd`, y determinar cuál es mayor.
   c) Reducir el intervalo de búsqueda, eliminando la parte que no contenga el valor mayor de `y`.
4. Evaluar `xmax` e `ymax`.
5. Escribir los valores de `xmax` e `ymax`, y parar.

Para traducir el esquema a un programa, primero se crea una función definida por el programador para evaluar la función matemática `y = x cos(x)`. Llamemos a esta función `curva`. Esta función se puede escribir fácilmente como se muestra a continuación:

```
/* evaluar la función y = x * cos(x) */

double curva(double x)

{
  return(x * cos(x));
}
```

Observe que `cos(x)` es una llamada a una función de biblioteca de C.

Ahora considérese el paso 3 del esquema de programa anterior, que realiza la reducción del intervalo. Este paso puede programarse como una función, que llamaremos `reducir`. Observe, sin embargo, que los valores representados por las variables `a`, `b`, `xi`, `xd`, `yi` e `yd`, que cambian en el curso del cómputo, deben ser transferidos a y desde esta función y `main`. Por tanto, haremos que estas variables sean externas y su ámbito incluya tanto a `reducir` como a `main`.

La función reducir se puede escribir como

```
/* rutina de reducción del intervalo */

void reducir(void)
{
  xi = a + 0.5 * (b - a - CNST);
  xd = xi + CNST;
```

```
   yi = curva(xi);
   yd = curva(xd);

   if (yi > yd)   {       /* retener el intervalo izquierdo */
      b = xd;
      return;
   }
   if (yi < yd)           /* retener el intervalo derecho */
      a = xi;
   return;
}
```

Observe que el parámetro al cual nos hemos referido antes como `sep` está representado ahora por la constante `CNST`. Observe también que esta función no incluye ningún argumento formal y no devuelve nada a través de la instrucción `return`. Toda la información transferida involucra variables externas.

Ahora es muy simple escribir la parte principal del programa, que llama a las dos funciones definidas anteriormente. Aquí está el programa completo.

```
/* encontrar el máximo de una función en un intervalo especificado */

#include <stdio.h>
#include <math.h>

#define CNST  0.0001

double a, b, xi, yi, xd, yd;    /* variables globales */

void reducir(void);             /* prototipo de función */
double curva(double xi);        /* prototipo de función */

main()

{
   double xmax, ymax;

   /* leer datos de entrada (puntos extremos del intervalo) */

   printf("\na = ");
   scanf("%lf", &a);
   printf("b = ");
   scanf("%lf", &b);

   /* bucle de reducción del intervalo */

   do
      reducir();
   while ((yi != yd) && ((b - a) > 3 * CNST));
```

```
    /* calcular xmax e ymax y escribir los resultados */

        xmax = 0.5 * (xi + xd);
        ymax = curva(xmax);
        printf("\nxmax = %8.6lf   ymax = %8.6lf", xmax, ymax);

}

/* rutina de reducción del intervalo */

void reducir(void)

{
   xi = a + 0.5 * (b - a - CNST);
   xd = xi + CNST;
   yi = curva(xi);
   yd = curva(xd);

   if (yi > yd)   {       /* retener el intervalo izquierdo */
      b = xd;
      return;
   }
   if (yi < yd)           /* retener el intervalo derecho */
      a = xi;
   return;
}
/* evaluar la función y = x * cos(x) */

double curva(double x)

{
    return(x * cos(x));
}
```

Las variables a, b, xi, yi, xd e yd están definidas como variables externas cuyo ámbito incluye todo el programa. Observe que estas variables están declaradas antes de que empiece main.

La ejecución del programa, con a = 0 y b = 3.141593, produce la siguiente sesión interactiva. Las respuestas del usuario se subrayan, como siempre.

```
a = 0
b = 3.141593

xmax = 0.860394   ymax = 0.561096
```

Por tanto, obtenemos la localización y el valor del máximo dentro del intervalo original dado.

A las variables externas se les pueden asignar valores iniciales como parte de la definición de variables, pero los valores iniciales se tienen que expresar como *constantes* en vez de como expresiones. Estos valores iniciales se asignarán sólo una vez, al comienzo del programa. Las variables externas retendrán estos valores iniciales, a menos que se alteren después durante la ejecución del programa.

Si no se incluye un valor inicial en la definición de la variable externa, se le asignará automáticamente el valor cero. Por tanto, las variables externas nunca se dejan bailando con valores iniciales indefinidos y vagos. No obstante, es una buena práctica de programación asignar un valor inicial explícito de cero cuando lo requiera la lógica del programa.

EJEMPLO 8.5. Longitud media de varias líneas de texto. A continuación se muestra una modificación del programa presentado en el Ejemplo 8.3, para determinar el número medio de caracteres en varias líneas de texto. La versión actual hace uso de variables externas para representar el número total (acumulado) de caracteres leídos y el número total de líneas.

```
/*leer varias líneas de texto y determinar
  el número medio de caracteres por línea */

#include <stdio.h>

int suma = 0;            /* número total de caracteres */
int lineas = 0;          /* número total de líneas */

int contlinea(void);

main()

{
   int n;                /* número de caracteres en una línea */
   float media;          /* número medio de caracteres por línea */

   printf("Introducir el texto debajo:\n");

   /* leer una línea de texto y actualizar los contadores */

   while ((n = contlinea()) > 0)  {
         suma += n;
         ++lineas;
   }

   media = (float) suma / lineas;
   printf("\nNúmero medio de caracteres por línea: %5.2f", media);
}

/* leer una línea de texto y contar el número de caracteres */

int contlinea(void)

{
   char linea[80];
   int cont = 0;

   while ((linea[cont] = getchar()) != '\n')
         ++cont;
   return (cont);
}
```

Observe que `suma` y `lineas` son variables externas que representan el número total (acumulado) de caracteres leídos y el número total de líneas, respectivamente. A ambas variables se les ha asignado el valor inicial cero. Estos valores se modifican sucesivamente dentro de `main`, según se leen líneas adicionales de texto.

Recuerde también que la versión anterior del programa usaba dos variables automáticas diferentes llamadas `cont` en partes diferentes del programa. En la versión actual, sin embargo, las variables que representan estas mismas cantidades tienen nombres distintos, pues una de las variables (`lineas`) es ahora una variable externa.

Debe destacarse que a `suma` y `lineas` no hay que asignarles explícitamente el valor cero, puesto que las variables externas siempre se inicializan a cero a menos que se designe algún otro valor. Se incluye el valor explícito de inicialización cero para clarificar la lógica del programa.

Los arrays pueden declararse como automáticos o como externos, si bien los arrays automáticos no se pueden inicializar. Veremos cómo se asignan valores iniciales a los elementos de un array en el Capítulo 9.

Finalmente, hay que destacar que existen peligros inherentes en el uso de variables externas, porque una alteración del valor de la variable externa dentro de una función repercutirá en otras partes del programa. A veces esto pasa inadvertidamente, como un *efecto lateral* de alguna otra acción. Por tanto, existe la posibilidad de que el valor de la variable externa cambie inesperadamente, dando lugar a un error sutil de programación. El programador ha de decidir cuidadosamente qué tipo de almacenamiento es más adecuado para cada situación de programación particular.

8.4. VARIABLES ESTÁTICAS

En esta sección y la siguiente se hace la distinción entre un programa *monoarchivo*, en el que todo el programa está contenido en un solo archivo fuente, y un programa *multiarchivo*, donde las funciones que componen el programa están almacenadas en archivos fuente separados. Las reglas que rigen el tipo de almacenamiento estático son diferentes en cada situación.

En un programa monoarchivo, las variables estáticas se definen dentro de funciones individuales y tienen, por tanto, el mismo ámbito que las variables automáticas; esto es, son locales a la función en que están definidas. Sin embargo, a diferencia de las variables automáticas, las variables estáticas retienen sus valores durante toda la vida del programa. Como consecuencia, si se sale de una función y posteriormente se vuelve a entrar, las variables estáticas definidas dentro de esa función retendrán sus valores previos. Esta característica permite a las funciones mantener información permanente a lo largo de toda la ejecución del programa.

Las variables estáticas se definen dentro de una función de la misma forma que las variables automáticas, excepto que la declaración de variables tiene que empezar con la designación del tipo de almacenamiento `static`. Las variables estáticas se pueden utilizar dentro de una función de la misma manera que las otras variables. Sin embargo, no se puede acceder a ellas fuera de la función en que están definidas.

No es inusual definir variables automáticas o estáticas con el mismo nombre que las variables externas. En tales situaciones, las variables locales tienen precedencia sobre las variables externas, aunque los valores de las variables externas no se verán afectados por la manipulación de las variables locales. Por tanto, las variables externas mantienen su independencia frente a las varia-

bles automáticas o estáticas. Lo mismo es cierto para variables locales dentro de una función que tienen los mismos nombres de variables locales dentro de otra función.

EJEMPLO 8.6. A continuación se muestra el esqueleto de la estructura de un programa en C que incluye variables pertenecientes a varios tipos diferentes de almacenamiento.

```
float a, b, c;

void ficticio(void);

main()
{
   static float a;

   . . . . .

}

void ficticio(void)
{
   static int a;
   int b;

   . . . . .

}
```

En este programa, `a`, `b` y `c` son variables externas en coma flotante. Sin embargo, `a` está redefinida como variable en coma flotante *estática* dentro de `main`. Por tanto, `b` y `c` son las únicas variables externas que se reconocerán dentro de `main`. Observe que la variable local *estática* `a` será independiente de la variable externa `a`.

De igual forma, `a` y `b` se redefinen como variables enteras dentro de `ficticio`. Observe que `a` es una variable estática, pero `b` es una variable automática. Luego `a` retendrá su valor previo si se vuelve a entrar a `ficticio`; sin embargo, `b` perderá su valor siempre que se transfiera el control fuera de `ficticio`. Además, `c` es la única variable *externa* que será reconocida dentro de `ficticio`.

Por ser `a` y `b` locales a `ficticio`, serán independientes de las variables externas `a`, `b` y `c`, y de la variable estática `a` definida dentro de `main`. El hecho de que `a` y `b` sean declaradas como variables enteras dentro de `ficticio` y como variables en coma flotante en cualquier otro sitio es irrelevante.

Se pueden incluir valores iniciales en las declaraciones de variables estáticas. Las reglas asociadas a la asignación de estos valores son esencialmente las mismas que las asociadas con la inicialización de variables externas, aunque las variables estáticas se definen localmente dentro de una función. En particular:

1. Los valores iniciales tienen que ser expresados como constantes, no como expresiones.
2. Los valores iniciales se asignan a sus respectivas variables al comienzo de la ejecución del programa. Las variables retienen estos valores a lo largo de toda la vida del programa, a menos que se asignen valores diferentes en el curso de la ejecución.
3. A todas las variables estáticas cuyas declaraciones no incluyan valores iniciales explícitos se les asignará el valor cero. Así las variables estáticas tendrán siempre valores asignados.

EJEMPLO 8.7. Generación de los números de Fibonacci. Los números de Fibonacci forman una interesante secuencia en la que cada número es igual a la suma de los dos números anteriores. En otras palabras:

$$F_i = F_{i-1} + F_{i-2}$$

donde F_i refiere el número *i*-ésimo de Fibonacci. Los dos primeros números son iguales a 1; esto es,

$$F_1 = F_2 = 1$$

Por tanto,

$$F_3 = F_2 + F_1 = 1 + 1 = 2$$
$$F_4 = F_3 + F_2 = 2 + 1 = 3$$
$$F_5 = F_4 + F_3 = 3 + 2 = 5$$

y así sucesivamente.

Escribamos un programa en C que genere los `n` primeros números de Fibonacci, donde `n` es un valor especificado por el usuario. En la parte `main` del programa se lee un valor para `n`, y luego se ejecuta un bucle que genera y escribe cada número de Fibonacci. Para calcular cada número de Fibonacci a partir de sus dos valores precedentes se utilizará una función llamada `fibonacci`. Esta función será llamada una vez en cada pasada por el bucle principal.

Cuando se entra en `fibonacci`, el cálculo del número actual de Fibonacci, *f*, es muy simple, supuestos conocidos los dos valores previos. Se pueden retener estos valores de una llamada a la función a la siguiente si se los asignamos a las variables estáticas `f1` y `f2`, que representan F_{i-1} y F_{i-2}, respectivamente. (Podríamos, por supuesto, haber usado variables externas para este propósito, pero es mejor usar variables locales, pues sólo se requieren F_{i-1} y F_{i-2} dentro de la función.) A continuación se calcula el número deseado de Fibonacci como

```
f = f1 + f2
```

y se actualizan los valores de `f2` y `f1` utilizando las fórmulas

```
f2 = f1
```

y

```
f1 = f
```

He aquí el programa completo en C.

```
/* programa para calcular números de Fibonacci sucesivos */

#include <stdio.h>

long int fibonacci(int cont);

main()
```

```
{
  int cont, n;

  printf("¿Cuantos números de Fibonacci? ");
  scanf("%d", &n);
  printf("\n");

  for (cont = 1; cont <= n; ++cont)
      printf("\ni = %2d    F = %ld", cont, fibonacci(cont));
}

long int fibonacci(int cont)

/* calcular un número de Fibonacci usando las fórmulas

  F = 1  para i < 3, y F = F1 + F2  para i >= 3 */

{
  static long int f1 = 1, f2 = 1;
  long int f;

  f = (cont < 3) ? 1 : f1 + f2;
  f2 = f1;
  f1 = f;
  return(f);
}
```

Observe que se han usado enteros largos para representar los números de Fibonacci. Observe también que `f1` y `f2` son variables estáticas de valor inicial `1`. Estos valores iniciales se asignan sólo una vez, al comienzo de la ejecución del programa. Los subsiguientes valores se retienen según son asignados, entre llamadas sucesivas a la función. Se sobreentiende que `f1` y `f2` son variables estrictamente locales, aunque retengan sus valores de una llamada a función a otra.

A continuación se muestra la salida correspondiente al valor `n = 30`. La respuesta del usuario está subrayada, como de costumbre.

```
¿Cuantos números de Fibonacci? 30

i =  1    F = 1
i =  2    F = 1
i =  3    F = 2
i =  4    F = 3
i =  5    F = 5
i =  6    F = 8
i =  7    F = 13
i =  8    F = 21
i =  9    F = 34
i = 10    F = 55
i = 11    F = 89
i = 12    F = 144
i = 13    F = 233
```

```
i = 14    F = 377
i = 15    F = 610
i = 16    F = 987
i = 17    F = 1597
i = 18    F = 2584
i = 19    F = 4181
i = 20    F = 6765
i = 21    F = 10946
i = 22    F = 17711
i = 23    F = 28657
i = 24    F = 46368
i = 25    F = 75025
i = 26    F = 121393
i = 27    F = 196418
i = 28    F = 317811
i = 29    F = 514229
i = 30    F = 832040
```

Es posible definir e inicializar arrays estáticas como sucede con variables estáticas simples. El uso de arrays se trata en el próximo capítulo.

8.5. PROGRAMAS DE VARIOS ARCHIVOS

Un *archivo* es una colección de información almacenada como una entidad separada dentro de la computadora o del dispositivo auxiliar de almacenamiento. Un archivo puede ser un conjunto de datos, un programa fuente, una parte de un programa fuente, un programa objeto, etc. En este capítulo consideraremos que un archivo es o un programa completo en C o una parte de un programa en C, por ejemplo una o más funciones. (Véase el Capítulo 12 para una discusión de los archivos de datos y su relación con los programas en C.)

Hasta ahora hemos restringido nuestra atención a programas en C que están enteramente contenidos en un archivo individual. Sin embargo, muchos programas están compuestos por varios archivos. Esto es especialmente cierto en programas que usan funciones grandes, donde cada función puede ocupar un archivo separado. O, si existen muchas funciones pequeñas relacionadas entre sí dentro de un programa, puede ser deseable colocar algunas funciones dentro de cada uno de los diversos archivos. Los archivos individuales se compilan de forma separada y a continuación se enlazan para formar un programa objeto ejecutable (ver sección 5.4). Esto hace más fácil la redacción y la depuración del programa, pues cada archivo se mantiene con un tamaño manejable.

Los programas multiarchivo permiten una mayor flexibilidad en la definición del ámbito de las funciones y de las variables. Sin embargo, las reglas asociadas con los tipos de almacenamiento se vuelven más complicadas, porque se aplican tanto a funciones como a variables, y se dispone de más opciones en el uso tanto de variables externas como estáticas.

Funciones

Empecemos considerando las reglas asociadas con la utilización de funciones. La definición de una función dentro de un programa multiarchivo puede ser tanto *externa* como *estática*. Una

función externa será reconocida a lo largo de todo el programa, mientras que una función estática será reconocida sólo dentro del archivo en el que se defina. En todo caso, el tipo de almacenamiento se establece colocando el tipo adecuado de asignación de tipo de almacenamiento (es decir, `extern` o `static`) al comienzo de la definición de la función. Se supone que la función es *externa* si no aparece la designación del tipo de almacenamiento.

En términos generales, la primera línea de una definición de función se puede escribir como

```
tipo-de-almacenamiento  tipo-datos  nombre(tipo 1  arg 1,
                    tipo 2  arg 2, . . ., tipo n  arg n)
```

donde *`tipo-de-almacenamiento`* denota el tipo de almacenamiento asociado con la función, *`tipo-datos`* el tipo de datos del valor devuelto por la función, *`nombre`* denota el nombre de la función, *`tipo 1, tipo 2, . . ., tipo n`* se refieren a los tipos de los argumentos formales y *`arg 1, arg 2, . . ., arg n`* a los argumentos formales. Recuerde que el tipo de almacenamiento, el tipo de datos y los argumentos formales no necesitan estar siempre presentes en cada definición de función.

Cuando se define una función en un archivo y se accede a ella en otro, este último archivo debe incluir una *declaración* de función. Esta declaración identifica la función como una función externa cuya definición aparece en otra parte. Estas declaraciones se colocan normalmente al comienzo del archivo, delante de cualquiera de las definiciones de funciones.

Es una buena práctica de programación empezar la declaración con un especificador de tipo de almacenamiento `extern`. Este especificador de tipo de almacenamiento no es absolutamente necesario, pues se supone que la función es externa si no está presente el especificador de tipo de almacenamiento.

En términos generales, una *declaración* de función se puede escribir como

```
tipo-de-almacenamiento tipo-datos  nombre(tipo argumento 1,
              tipo argumento 2, . . ., tipo argumento n);
```

Una declaración de función también se puede escribir utilizando el prototipo completo de función (ver sección 7.4) como

```
tipo-de-almacenamiento  tipo-datos  nombre(tipo 1  arg 1,
                    tipo 2  arg2, . . ., tipo n  arg n);
```

Recuerde que el tipo de almacenamiento, el tipo de datos y los tipos de argumentos no necesitan estar presentes en cada declaración de función.

Para ejecutar un programa multiarchivo, se debe compilar cada archivo individual, y los archivos objeto resultantes se deben enlazar. Para hacer esto, normalmente se combinan los archivos fuente en un *proyecto*. A continuación se *construye* («*build*») el proyecto (es decir, se compilan todos los archivos fuente y se enlazan los archivos objeto resultantes en un único programa ejecutable). Si más adelante se modifica algunos de los archivos fuente, *construimos* («*make*») otro programa ejecutable (es decir, compilamos los nuevos archivos fuente y enlazamos los archivos objeto resultantes, con los archivos objeto que no se han modificado, en un nuevo programa ejecutable). Los detalles de cómo se efectúa esto cambian de una versión de C a otra.

EJEMPLO 8.8. Aquí se muestra un programa simple que genera el mensaje «¡Hola!» desde una función. El programa consta de dos funciones: `main` y `salida`. Cada función aparece en un archivo separado.

Primer archivo:

```
/* programa simple multiarchivo para escribir "hola" */

#include <stdio.h>

extern void salida(void);     /* prototipo de función */

main()
{
   salida();
}
```

Segundo archivo:

```
extern void salida(void)      /* definición de función externa */
{
   printf("¡Hola!");
   return;
}
```

Observe que a `salida` se le asigna el tipo de almacenamiento `extern`, pues se tiene que acceder a ella desde otro archivo distinto de aquel en el que está definida; por tanto, ha de ser una función externa. Así se incluye la palabra reservada `extern` tanto en la declaración de la función (en el primer archivo) como en la definición de la función (segundo archivo). Como `extern` es el tipo de almacenamiento por defecto, se puede omitir la palabra reservada `extern` tanto de la declaración como de la definición de función. De este modo, el programa se puede escribir como sigue:

Primer archivo:

```
/* programa simple multiarchivo para escribir "hola" */

#include <stdio.h>

void salida(void);            /* prototipo de función */

main()
{
   salida();
}
```

Segundo archivo:

```
void salida(void)   /* definición de función externa */

{
   printf("¡Hola!");
   return;
}
```

Construyamos un proyecto Turbo C++ correspondiente a este programa multiarchivo. Para hacerlo, introducimos en primer lugar el código fuente del primer archivo y lo guardamos en un archivo llamado

ARCHIVO1.C. A continuación introducimos el código fuente del segundo archivo y lo guardamos en un archivo llamado ARCHIVO2.C. Estos dos archivos se muestran en la Figura 8.4 en sendas ventanas.

A continuación seleccionamos la opción New del menú Project, y especificamos como nombre del proyecto EX8-8.IDE. Como resultado, se abre la ventana Project, aproximadamente en el centro de la Figura 8.4. En esta ventana vemos que el proyecto generará un programa ejecutable llamado EX8-8.EXE. Este programa ejecutable se obtiene a partir de los dos archivos fuente, ARCHIVO1.C y ARCHIVO2.C.

El programa ya se puede ejecutar seleccionando Run en el menú Debug, como se explicó en el Capítulo 5 (ver Ejemplo 5.4). En la ventana de salida se visualiza el mensaje resultante, ¡Hola!, tal y como muestra la parte inferior derecha de la Figura 8.4.

Si un archivo contiene una función estática, será necesario incluir el tipo de almacenamiento static en la declaración o prototipo de la función.

EJEMPLO 8.9. Simulación de un juego de azar (juego de dados «Craps»). Ésta es otra versión del juego de dados «Craps», presentado originalmente en el Ejemplo 7.11. En esta versión el programa consta de dos archivos separados. El primer archivo contiene la función main, mientras que el segundo contiene las funciones juego y tirada.

Primer archivo:

```
/* simulación del juego de dados "Craps" */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
```

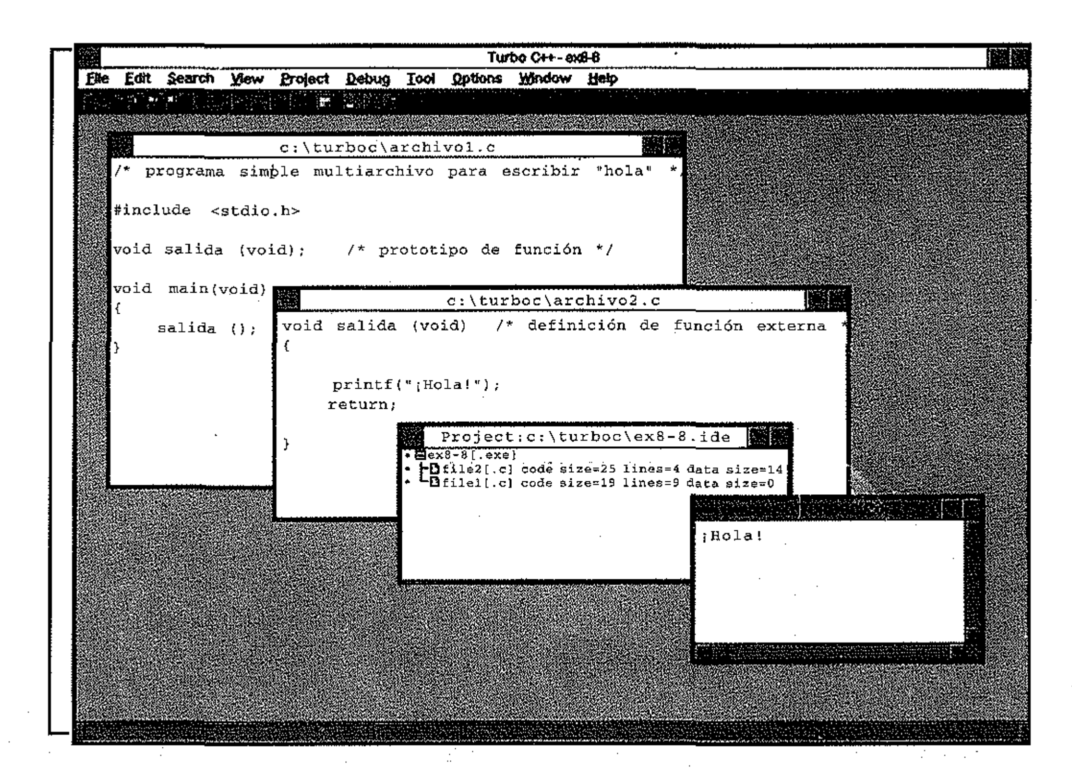


Figura 8.4.

```
#define SEMILLA   12345

extern void juego(void);    /* prototipo de función */

main()
{
  char respuesta = 'S';

  printf("Bienvenido al juego CRAPS\n\n");
  printf("Para lanzar los dados, pulsa Intro\n\n");

  srand(SEMILLA);  /* inicializa el generador de números aleatorios */

  /* bucle principal */

  while (toupper(respuesta) != 'N')    {
       juego();
       printf("\n¿Deseas jugar de nuevo? (S/N) ");
       scanf(" %c", &respuesta);
       printf("\n");
  }
  printf("Adiós, que lo pases bien");
}
```

Segundo archivo:

```
#include <stdio.h>
#include <stdlib.h>

static int tirada(void);     /* prototipo de función */
extern void juego(void)      /* definición de función externa */

/* simular una jugada completa */

{
  int puntos1, puntos2;
  char nada;

  printf("\nPor favor lanza los dados . . .");
  scanf("%c", &nada);
  printf("\n");
  puntos1 = tirada();
  printf("\n%2d", puntos1);

  switch (puntos1)   {

  case 7:   /* se ha ganado en la primera tirada */
  case 11:

       printf(" - ¡Felicidades! GANASTE en la primera tirada\n");
       break;
```

```
   case 2:    /* se ha perdido en la primera tirada */
   case 3:
   case 12:

        printf(" - ¡Lo siento! - PERDISTE en la primera tirada\n");
        break;

   case 4:    /* se requieren otras tiradas */
   case 5:
   case 6:
   case 8:
   case 9:
   case 10:

        do    {
                printf(" - Lanza los dados de nuevo . . .");
                scanf("%c", &nada);
                puntos2 = tirada();
                printf("\n%2d", puntos2);
        } while (puntos2 != puntos1 && puntos2 != 7);

        if (puntos2 == puntos1)
           printf(" - GANAS por igualar tu primera puntuación\n");
        else
          printf(" - PIERDES por no igualar tu primera puntuación\n");
        break;
   }

   return;
}

/* simula el lanzamiento de un par de dados */

static int tirada(void) /* definición de función estática */

{
   float x1, x2;          /* números en coma flotante aleatorios entre
                             0 y 1 */
   int n1, n2;            /* números enteros aleatorios entre 1 y 6 */

   x1 = rand() / 32768.0;
   x2 = rand() / 32768.0;

   n1 = 1 + (int) (6 * x1);  /* simula primer dado */
   n2 = 1 + (int) (6 * x2);  /* simula segundo dado */

   return(n1 + n2);       /* la puntuación es la suma de los dos dados */
}
```

Observe que `juego` se define como una función externa, de modo que se puede acceder desde `main` (porque `main` y `juego` se definen en archivos separados). Por tanto, `juego` está declarada como una

función externa en el primer archivo. Por otro lado, a `tirada` sólo se accede desde `juego`. Tanto `tirada` como `juego` se definen en el segundo archivo. Así pues, `tirada` no necesita ser reconocida en el primer archivo. Podemos definir `tirada` como función estática, confinando su ámbito al segundo archivo.

Observe también que cada archivo tiene un conjunto separado de instrucciones `#include` para los archivos de cabecera `stdio.h` y `stdlib.h`. Esto asegura que en cada archivo se incluyan las declaraciones necesarias para las funciones de biblioteca.

Cuando se compilan y enlazan los archivos individuales, y se ejecuta el programa ejecutable resultante, el programa genera un diálogo idéntico al mostrado en el Ejemplo 7.11, como era de esperar.

Variables

En un programa multiarchivo se pueden definir variables externas (globales) en un archivo y acceder a ellas en otro. De nuevo enfatizamos la distinción entre la *definición* de una variable externa y sus *declaraciones*. Una *definición* de variable externa puede aparecer sólo en un archivo. Su localización dentro del archivo debe ser externa a cualquier definición de función. Generalmente aparecerá al comienzo del archivo, delante de la primera definición de función.

Las definiciones de variables externas pueden incluir valores iniciales. Cualquier variable externa que no tenga asignado un valor inicial se inicializará automáticamente a cero. No es necesario el especificador de tipo de almacenamiento `extern` dentro de la definición; de hecho, muchas versiones de C prohíben específicamente la aparición de este especificador de tipo de almacenamiento en *definiciones* de variables externas. Las definiciones de variables externas se reconocen por su localización en los archivos en que están definidas y en los que aparecen. Seguiremos este convenio en este libro.

Para acceder a una variable externa en otro archivo, se debe *declarar* primero en ese archivo. Esta declaración puede aparecer en cualquier parte dentro del archivo. Sin embargo, generalmente se colocará al principio del archivo, delante de la primera definición de función. La declaración *debe* comenzar con el especificador de tipo de almacenamiento `extern`. *No pueden* incluirse valores iniciales en las declaraciones de variables externas.

El valor asignado a una variable externa puede alterarse dentro de cualquier archivo donde se reconozca la variable. Estos cambios serán reconocidos en todos los otros archivos que estén dentro del ámbito de la variable. Por ello, las variables externas proporcionan los medios adecuados para transferir información entre archivos.

EJEMPLO 8.10. A continuación se muestra el esquema de un programa en C de dos archivos que utiliza variables externas.

<u>*Primer archivo:*</u>

```
int a = 1, b = 2, c = 3;      /* DEFINICION de variable externa */

extern void func1(void);      /* DECLARACION de función externa */

main()                        /* DEFINICION de función */
{
    . . . . .
}
```

Segundo archivo:

```
extern int a, b, c;          /* DECLARACION de variables externas */

extern void func1(void)      /* DEFINICION de función externa */
{
     . . . . .
}
```

Las variables a, b y c se definen como externas en el primer archivo, y se les asigna 1, 2 y 3 como valor inicial, respectivamente. El primer archivo también contiene una *definición* de la función main y una *declaración* para la función externa func1, definida en otro lugar. Dentro del segundo archivo vemos la *definición* de func1 y una *declaración* de las variables externas a, b y c.

Observe que el especificador de tipo de almacenamiento extern aparece tanto en la *definición* como en la *declaración* de la función externa func1. Este especificador de tipo de almacenamiento está también presente en la *declaración* de las *variables* externas (en el segundo archivo), pero *no* aparece en la *definición* de las variables externas (en el primer archivo).

El ámbito de a, b y c es el programa completo. Por tanto, se puede acceder a estas variables y a sus valores en ambos archivos, tanto en main como en func1.

EJEMPLO 8.11. Búsqueda de un máximo. En el Ejemplo 8.4 se presentaba un programa en C que determinaba el valor de x para el cual la función

$$y = x \cos(x)$$

se hacía máxima en un intervalo especificado. Ahora se presenta otra versión de este programa, donde cada una de las tres funciones que componen el programa está situada en un archivo separado.

Primer archivo:

```
/* encontrar el máximo de una función en un intervalo especificado */

#include <stdio.h>

double a, b, xi, yi, xd, yd, cnst = 0.0001;
                                /* definición de variables externas */

extern void reducir(void);      /* prototipo de función externa */

extern double curva(double xi);
                                /* prototipo de función externa */

main()                          /* definición de función */
{
  double xmax, ymax;

  /* leer datos de entrada (puntos extremos del intervalo) */

  printf("\na = ");
  scanf("%lf", &a);
```

```
   printf("b = ");
   scanf("%lf", &b);

   /* bucle de reducción del intervalo */

   do
      reducir();
   while ((yi != yd) && ((b - a) > 3 * cnst));

   /* calcular xmax e ymax y escribir los resultados */

   xmax = 0.5 * (xi + xd);
   ymax = curva(xmax);
   printf("\nxmax = %8.6lf   ymax = %8.6lf", xmax, ymax);
}
```

Segundo archivo:

```
/* rutina de reducción del intervalo */

extern double a, b, xi, yi, xd, yd, cnst;
                              /* declaración  de variables externas */

extern double curva(double xi);
                              /* prototipo de función externa */

extern void reducir(void)  /* definición de función externa */

{
   xi = a + 0.5 * (b - a - cnst);
   xd = xi + cnst;
   yi = curva(xi);
   yd = curva(xd);

   if (yi > yd)    {   /* retener el intervalo izquierdo */
      b = xd;
      return;
}
   if (yi < yd)        /* retener el intervalo derecho */
      a = xi;
   return;

}
```

Tercer archivo:

```
/* evaluar la función y = x * cos(x) */

#include <math.h>

extern double curva(double x)      /* definición de función externa */

{
   return(x * cos(x));
}
```

La función externa `reducir`, definida en el segundo archivo, está declarada en el primer archivo. Por tanto, su ámbito cubre los dos primeros archivos. Igualmente, la función externa `curva`, definida en el tercer archivo, se declara en los archivos primero y segundo. Luego su ámbito es todo el programa. Observe que el especificador de tipo de almacenamiento `extern` aparece tanto en las definiciones de función como en los prototipos de funciones.

Consideremos ahora las variables externas `a`, `b`, `xi`, `yi`, `xd`, `yd` y `cnst`, definidas en el primer archivo. Observe que a `cnst` se le asigna valor inicial en la definición. Estas variables se utilizan, y por tanto se declaran, en el segundo archivo, pero no en el tercero. Observe que la *declaración* de variables (en el segundo archivo) incluye el especificador de tipo de almacenamiento `extern`, pero en la *definición* de variables (en el primer archivo) no se incluye especificador de tipo de almacenamiento.

Finalmente, observe la instrucción `#include <math.h>` al comienzo del tercer archivo. Esta instrucción hace que el archivo de cabecera `math.h` se incluya en el tercer archivo fuente, dando soporte a la función de biblioteca `cos`.

Los resultados de la ejecución de este programa son idénticos a los mostrados en el Ejemplo 8.4.

En un archivo, las variables externas se pueden definir como estáticas. Para hacer esto se coloca el especificador de tipo de almacenamiento `static` al principio de la definición. El ámbito de una variable externa estática será la parte restante del archivo en el cual ha sido definida. No será reconocida en ninguna otra parte del programa (es decir, en otros archivos). El uso de variables externas estáticas dentro de un archivo permite que un grupo de variables esté «oculto» al resto del programa. Otras variables externas pueden estar definidas con el mismo nombre en los archivos restantes. (Sin embargo, generalmente no es una buena idea utilizar nombres idénticos de variables. Estas variables con el mismo nombre pueden causar confusión en la comprensión de la lógica del programa, aunque no entrarán en conflicto sintáctico unas con otras).

EJEMPLO 8.12. Generación de números de Fibonacci. Volvamos al problema de calcular números de Fibonacci, que originalmente consideramos en el Ejemplo 8.7. Si se reescribe el programa como un programa de dos archivos utilizando variables estáticas externas, se obtiene el siguiente programa completo.

Primer archivo:

```
/* programa para calcular números de Fibonacci sucesivos */

#include <stdio.h>

extern long int fibonacci(int cont); /* prototipo de función exter-
                                        na */

main()                                /* definición de función */
{
   int cont, n;

   printf("¿Cuantos números de Fibonacci? ");
   scanf("%d", &n);
   printf("\n");
```

```
    for (cont = 1; cont <= n; ++cont)
        printf("\ni = %2d   F = %ld", cont, fibonacci(cont));
}
```

Segundo archivo:

```
/* calcular un número de Fibonacci (F=1 para i<3, y F=F1+F2 para i>=3 */

static long int f1 = 1, f2 = 1;  /* definición de variables exter-
                                     nas estáticas */

long int fibonacci(int cont)     /* definición de función externa */
{
    long int f;

    f = (cont < 3) ? 1 : f1 + f2;
    f2 = f1;
    f1 = f;
    return(f);
}
```

En este programa la función `fibonacci` se define en el segundo archivo y se declara en el primero; por ello su ámbito es todo el programa. Por otro lado, las variables `f1` y `f2` se definen como variables externas estáticas. Obsérvese que la definición de variables en el segundo archivo incluye la asignación de valores iniciales.

Los resultados que se obtienen son idénticos a los mostrados en el Ejemplo 8.7.

8.6. MÁS SOBRE FUNCIONES DE BIBLIOTECA

Nuestra discusión sobre programas multiarchivo puede aportar una visión adicional sobre el uso de funciones de biblioteca. Recuerde que las funciones de biblioteca son rutinas preescritas que realizan operaciones o cálculos que se usan comúnmente (ver sección 3.6). Están contenidas en uno o varios archivos de biblioteca que acompañan a cada compilador de C.

Durante el proceso de convertir un programa fuente de C en programa objeto ejecutable, el programa fuente compilado se enlaza con los *archivos de biblioteca* para producir el programa ejecutable final. De esta forma, el programa final se construirá a partir de varios archivos, aunque el programa fuente original esté contenido en un solo archivo. El programa fuente debe, por tanto, incluir las declaraciones para las funciones de biblioteca, lo mismo que para las funciones definidas por el programador que se definen en archivos separados.

Un modo de proporcionar las funciones de biblioteca necesarias consiste en que el programador las escriba explícitamente, como en los programas multiarchivo mostrados en la última sección. Esto puede resultar tedioso, pues un programa pequeño puede hacer uso de varias funciones de biblioteca. Deseamos simplificar al máximo el uso de funciones de biblioteca. C ofrece una forma inteligente de hacerlo, al colocar las declaraciones de funciones de biblioteca necesarias en archivos fuente especiales llamados *archivos de cabecera.*

La mayoría de los compiladores de C incluyen varios archivos de cabecera, cada uno de los cuales contiene declaraciones funcionalmente relacionadas (ver Apéndice H). Por ejemplo, `stdio.h` es un archivo de cabecera que contiene declaraciones para rutinas de entrada/salida; `math.h` contiene declaraciones para ciertas funciones matemáticas; y así sucesivamente. Los

archivos de cabecera también contienen otra información relacionada con el uso de funciones de biblioteca, tal como las definiciones de constantes simbólicas.

Los archivos de cabecera deben ser combinados con el programa fuente durante el proceso de compilación. Esto se logra colocando una o más instrucciones `#include` al comienzo del programa fuente (o al comienzo de los archivos individuales del programa). Hemos estado siguiendo este procedimiento en todos los ejemplos de programación mostrados en este libro.

EJEMPLO 8.13. Interés compuesto. El Ejemplo 5.2 presentaba originalmente el siguiente programa C para efectuar cálculos simples de interés compuesto.

```
/* problema sencillo de interés compuesto */

#include <stdio.h>
#include <math.h>

main()

{
   float p, r, n, i, f;

   /* leer datos de entrada (mediante peticiones rotuladas) */

   printf("Por favor, introduce la suma inicial (P): ");
   scanf("%f", &p);
   printf("Por favor, introduce el interés (r): ");
   scanf("%f", &r);
   printf("Por favor, introduce el número de años (n): ");
   scanf("%f", &n);

   /* calcular i y f */

   i = r/100;
   f = p * pow((1 + i), n);

   /* escribir la salida */

   printf("\nEl valor final (F) es: %.2f\n", f);
}
```

Este programa usa dos archivos de cabecera, `stdio.h` y `math.h`. El primer archivo de cabecera contiene declaraciones para las funciones `printf` y `scanf`, mientras que el segundo contiene declaraciones para la función potenciación, `pow`.

Se puede reescribir este programa quitando las instrucciones `#include` y añadiendo nuestras propias declaraciones de funciones como sigue.

```
/* problema sencillo de interés compuesto */

extern int printf();  /* declaración de función de biblioteca */
extern int scanf();   /* declaración de función de biblioteca */
extern double pow(double, double);  /* declaración de función de bi-
                                       blioteca */

main()

{
   float p, r, n, i, f;
```

```
    /* leer datos de entrada (mediante peticiones rotuladas) */

    printf("Por favor, introduce la suma inicial (P): ");
    scanf("%f", &p);
    printf("Por favor, introduce el interés (r): ");
    scanf("%f", &r);
    printf("Por favor, introduce el número de años (n): ");
    scanf("%f", &n);

    /* calcular i y f */

    i = r/100;
    f = p * pow((1 + i), n);

    /* escribir la salida */

    printf("\nEl valor final (F) es: %.2f\n", f);
}
```

Esta versión del programa se compila de la misma manera que la versión anterior y generará la misma salida cuando se ejecute. En la práctica no se hace uso de estas declaraciones de funciones suplementarias definidas por el programador, pues es más complicado y generan nuevas fuentes de error. Es más, el proceso de comprobación de errores que se realiza durante el proceso de compilación es menos completo, porque no se especifican los tipos de argumentos para las funciones `printf` y `scanf`. (Observe que el número de argumentos en `printf` y `scanf` puede variar de una llamada de función a otra. La manera en que se especifican los tipos de argumentos bajo esas condiciones va más allá del ámbito de esta discusión.)

La *independencia de la plataforma* (es decir, la *independencia de la máquina*) es una ventaja significativa en este acercamiento al uso de funciones de biblioteca y de archivos de cabecera. De este modo, las características dependientes de la máquina pueden proporcionarse como funciones de biblioteca, como constantes de caracteres o como *macros* (ver sección 14.4) incluidas en los archivos de cabecera. Un programa típico en C funcionará en muchas computadoras diferentes sin modificaciones, siempre que se usen las funciones de biblioteca y los archivos de cabecera apropiados. La portabilidad resultante de este enfoque es la mayor contribución a la popularidad del C.

CUESTIONES DE REPASO

8.1. ¿Qué significado tiene el tipo de almacenamiento de una variable?
8.2. Nombrar las cuatro especificaciones de tipos de almacenamiento incluidas en C.
8.3. ¿Qué significa el ámbito de una variable dentro de un programa?
8.4. ¿Cuál es la finalidad de una variable automática? ¿Cuál es su ámbito?
8.5. ¿Cómo se define una variable automática? ¿Cómo se inicializa? ¿Qué sucede si una variable automática no se inicializa explícitamente dentro de una función?
8.6. ¿Retiene una variable automática su valor una vez que el control es transferido fuera de la función en la que se define?
8.7. ¿Cuál es la finalidad de una variable externa? ¿Cuál es su ámbito?
8.8. Mencionar las diferencias entre una definición de variable externa y una declaración de variable externa.

8.9. ¿Cómo se define una variable externa? ¿Cómo se inicializa? ¿Qué sucede si una definición de variable externa no incluye la asignación de un valor inicial? Comparar las respuestas con las de variables automáticas.

8.10. Suponer que una variable externa se define fuera de una función `A` y se accede a ella dentro de ésta. ¿Importa si la variable externa se define antes o después de la función? Explicarlo.

8.11. ¿En qué sentido es más restrictiva la inicialización de una variable externa que la de una variable automática?

8.12. ¿Qué se entiende por efectos laterales?

8.13. ¿Qué peligros inherentes hay en el uso de variables externas?

8.14. ¿Cuál es la finalidad de una variable estática en un programa de un solo archivo? ¿Cuál es su ámbito?

8.15. ¿Cómo se define una variable estática en un programa de un solo archivo? ¿Cómo se inicializa una variable estática? Comparar con las variables automáticas.

8.16. ¿Bajo qué circunstancias puede ser deseable tener un programa compuesto por varios archivos?

8.17. Comparar la definición de funciones dentro de un programa multiarchivo con la definición de funciones dentro de un programa de un solo archivo. ¿Qué opciones adicionales están disponibles en el caso multiarchivo?

8.18. En un programa multiarchivo, ¿cuál es el tipo de almacenamiento por omisión para una función si no se incluye explícitamente en la definición?

8.19. ¿Cuál es la finalidad de una función estática en un programa multiarchivo?

8.20. Comparar la definición de variables externas en un programa multiarchivo con la definición de variables externas en un programa de un solo archivo. ¿Qué opciones adicionales están disponibles en el caso multiarchivo?

8.21. Comparar las definiciones de variables externas con las declaraciones de variables externas en un programa multiarchivo? ¿Cuál es la finalidad de cada una? ¿Puede una declaración de variable externa incluir la asignación de un valor inicial?

8.22. ¿Bajo qué circunstancias puede definirse una variable externa como estática? ¿Qué ventaja puede haber en hacer esto?

8.23. ¿Cuál es el ámbito de una variable estática externa?

8.24. ¿Cuál es la finalidad de un archivo de cabecera? ¿Es absolutamente necesario el uso de un archivo de cabecera?

PROBLEMAS

8.25. Describir la salida generada por cada uno de los siguientes programas.

a)
```
#include <stdio.h>

int func1(int cont);

main()
{
```

```
  int a, cont;

  for (cont = 1; cont <= 5; ++cont)  {
      a = func1(cont);
      printf("%d  ", a);
  }
}

func1(int x)
{
  int y = 0;

  y += x;
  return(y);
}
```

b)
```
#include <stdio.h>

int func1(int cont);

main()
{
  int a, cont;

  for (cont = 1; cont <= 5; ++cont)  {
      a = func1(cont);
      printf("%d  ", a);
  }
}

func1(int x)
{
  static int y = 0;

  y += x;
  return(y);
}
```

c)
```
#include <stdio.h>

int func1(int a);
int func2(int a);

main()
{
  int a = 0, b = 1, cont;

  for (cont = 1; cont <= 5; ++cont)  {
      b += func1(a) + func2(a);
      printf("%d  ", b);
  }
}
```

```
func1(int a)
{
   int b;

   b = func2(a);
   return(b);
}

func2(int a)
{
   static int b = 1;

   b += 1;
   return(b + a);
}
```

8.26. Escribir la primera línea de la definición de función para cada una de las siguientes situaciones descritas a continuación.

a) El segundo archivo de un programa con dos archivos contiene una función llamada resolver que acepta dos cantidades en coma flotante y devuelve un argumento en coma flotante. La función será llamada por otras funciones que están definidas en ambos archivos.

b) El segundo archivo de un programa con dos archivos contiene una función llamada resolver que acepta dos cantidades en coma flotante y devuelve un argumento en coma flotante, como en el problema anterior. El reconocimiento de esta función debe ser local al segundo archivo.

8.27. Añadir las declaraciones de funciones requeridas (o sugeridas) para cada uno de los esquemas mostrados a continuación.

a) Éste es un programa de dos archivos.

Primer archivo:

```
main()
{
   double x, y, z;

   . . . . .

   z = func1(x, y);

   . . . . .
}
```

Segundo archivo:

```
double func1(double a, double b)
{
   . . . . .
}
```

b) Éste es un programa de dos archivos.

Primer archivo:

```
main()
{
   double x, y, z;

   . . . . .

   z = func1(x, y);

   . . . . .
}
```

Segundo archivo:

```
double func1(double a, double b)
{
   double c;

   c = func2(a, b);
   . . . . .

}
static double func2(double a, double b)
{

   . . . . .

}
```

8.28. Describir la salida generada por cada uno de los siguientes programas.

a)
```
#include <stdio.h>

int a = 3;

int func1(int cont);

main()
{
   int cont;

   for (cont = 1; cont <= 5; ++cont)  {
       a = func1(cont);
       printf("%d  ", a);
   }
}
```

```
func1(int x)
{
  a += x;
  return(a);
}
```

b)
```
#include <stdio.h>

int a = 100, b = 200;

int func1(int a, int b);

main()
{
  int cont, c, d;

  for (cont = 1; cont <= 5; ++cont)  {
      c = 20 * (cont - 1);
      d = 4 * cont * cont;
      printf("%d  %d  ", func1(a, c), func1(b, d));
  }
}

func1(int x, int y)
{
  return(x - y);
}
```

c)
```
#include <stdio.h>

int a = 100, b = 200;

int func1(int c);

main()
{
  int cont, c;

  for (cont = 1; cont <= 5; ++cont)  {
      c = 4 * cont * cont;
      printf("%d  ", func1(c));
  }
}

func1(int x)
{
  int c;

  c = (x < 50) ? (a + x) : (b - x);
  return(c);
}
```

d)
```
#include <stdio.h>

int a = 100, b = 200;

int func1(int cont);
int func2(int c);

main()
{
   int cont;

   for (cont = 1; cont <= 5; ++cont)
        printf("%d  ", func1(cont));
}

func1(int x)
{
   int c, d;

   c = func2(x);
   d = (c < 100) ? (a + c) : b;
   return(d);
}

func2(int x)
{
   static int prod = 1;

   prod *= x;
   return(prod);
}
```

e)
```
#include <stdio.h>

int func1(int a);
int func2(int b);

main()
{
   int a = 0, b = 1, cont;

   for (cont = 1; cont <= 5; ++cont)  {
        b += func1(a + 1) + 1;
        printf("%d  ", b);
   }
}

func1(int a)
{
   int b;
```

```
    b = func2(a + 1) + 1;
    return(b);
}

func2(int a)
{
    return(a + 1);
}
```

f)

```
#include <stdio.h>

int a = 0, b = 1;

int func1(int a);
int func2(int b);

main()
{
    int cont;

    for (cont = 1; cont <= 5; ++cont)  {
        b += func1(a + 1) + 1;
        printf("%d  ", b);
    }
}

func1(int a)
{
    int b;

    b = func2(a + 1) + 1;
    return(b);
}

func2(int a)
{
    return(a + 1);
}
```

g)

```
#include <stdio.h>

int a = 0, b = 1;

int func1(int a);
int func2(int b);

main()
{
    int cont;
```

```
    for (cont = 1; cont <= 5; ++cont)   {
        b += func1(a + 1) + 1;
        printf("%d  ", b);
    }
}

func1(int a)
{
    b = func2(a + 1) + 1;
    return(b);
}

func2(int a)
{
    return(b + a);
}
```

h)
```
#include <stdio.h>

int cont = 0;

void func1(void);

main()
{
    printf("Por favor, introduzca una línea de texto\n");
    func1();
    printf("%d", cont);
}

void func1(void)
{
    char c;

    if ((c = getchar()) != '\n')    {
        ++cont;
        func1();
    }
    return;
}
```

PROBLEMAS DE PROGRAMACIÓN

8.29. El programa dado en el Ejemplo 8.4 puede modificarse fácilmente para *minimizar* una función de x. Este procedimiento de minimización nos puede proporcionar una técnica altamente eficaz para calcular las raíces de una ecuación algebraica no lineal. Por ejemplo, supóngase que queremos

encontrar el valor concreto de x que hace que una función $f(x)$ sea igual a cero. Una función típica de este tipo puede ser:

$$f(x) = x + \cos(x) - 1 - \text{sen}(x)$$

Si hacemos $y(x) = f(x)^2$, entonces la función $y(x)$ será siempre positiva, excepto para aquellos valores de x que sean raíces de la función dada [para los cuales $f(x)$ y por tanto $y(x)$ son iguales a cero]. Por tanto, cualquier valor de x que minimice a $y(x)$ es una raíz de la ecuación $f(x) = 0$.

Modificar el programa del Ejemplo 8.4 para minimizar una función dada. Usar el programa para obtener las raíces de las siguientes ecuaciones:

a) $x + \cos(x) = 1 + \text{sen}(x)$, $\pi/2 < x < \pi$
b) $x^5 + 3x^2 = 10$, $0 <= x <= 3$ (ver Ejemplo 6.21)

8.30. Modificar el programa del Ejemplo 7.11 de forma que se simule automática y no interactivamente una secuencia del juego de dados «Craps». Introducir el número total de juegos como variable de entrada. Incluir dentro del programa un contador que determine el número total de victorias. Utilizar el programa para simular un número grande de juegos (por ejemplo 1000). Estimar la probabilidad de ganar en este juego de dados. Este valor, expresado en decimal, es igual al número de victorias dividido entre el número total de partidas jugadas. Si la probabilidad excede de 0,500, es favorable al jugador; de lo contrario es favorable a la banca.

8.31. Reescribir cada uno de los siguientes programas de forma que incluya al menos una función definida por el programador, además de la función principal. Tenga cuidado con la elección de argumentos y (si es necesario) de variables externas.

a) Calcular la media ponderada de una lista de números [ver Problema 6.69(*a*)].
b) Calcular el producto acumulado de una lista de números [ver Problema 6.69(*b*)].
c) Calcular la media geométrica de una lista de números [ver Problema 6.69(*c*)].
d) Calcular y tabular una lista de números primos [ver Problema 6.69(*f*)].
e) Calcular el seno de x, usando el método descrito en el Problema 6.69(*i*).
f) Calcular los pagos de un préstamo [ver Problema 6.69(*j*)].
g) Determinar la calificación media en los exámenes de cada estudiante de la clase, como se describe en el Problema 6.69(*k*).

8.32. Escribir un programa completo en C para resolver cada uno de los problemas descritos a continuación. Utilizar funciones definidas por el programador donde corresponda. Compilar y ejecutar el programa con los datos dados en la descripción del problema.

a) Suponga que se deposita una cantidad dada de dinero A en una libreta de ahorros al principio de cada año durante n años. Si se percibe un interés i anual, entonces la cantidad F de dinero que se acumulará tras n años viene dada por

$$F = A\,[(1 + i/100) + (1 + i/100)^2 + (i + i/100)^3 + \ldots + (1 + i/100)^n]$$

Escribir un programa C en estilo conversacional que determine lo siguiente:

i) ¿Cuánto dinero se acumulará después de 30 años si se depositan 100 dólares al comienzo de cada año y el tipo de interés compuesto es del 6 por 100 anual?
ii) ¿Cuánto dinero se debe de depositar al comienzo de cada año para acumular 100000 dólares después de 30 años, suponiendo también un tipo de interés compuesto del 6 por 100 anual?

En cada caso, determinar primero la cantidad de dinero desconocida. Luego crear una tabla mostrando la cantidad total de dinero que se acumulará al final de cada año. Utilizar la función escrita para el Problema 7.43 para obtener la potenciación.

b) Modificar el programa anterior para obtener intereses trimestrales en vez de anuales. Comparar los resultados obtenidos para ambos problemas. *Sugerencia:* la fórmula adecuada es

$$F = A\,[(1 + i/100m)^{m} + (1 + i/100m)^{2m} + (i + i/100m)^{3m} + \ldots + (1 + i/100m)^{nm}]$$

donde m representa el número de períodos de interés por año.

c) El coste de la hipoteca de una casa se determina de modo que el deudor paga la misma cantidad de dinero al mes a la institución prestataria durante el tiempo de hipoteca. La fracción del pago mensual total que se destina a los intereses del total principal del préstamo varía, sin embargo, de un mes a otro. Al comienzo de la hipoteca la mayoría del pago mensual se destina al pago de los intereses y sólo una pequeña fracción a reducir el pago del préstamo. Gradualmente, el principal del préstamo se reduce, lo que provoca que el pago mensual de interés decrezca. Como consecuencia, el total del préstamo se reduce a ritmo acelerado.

Normalmente, los compradores de casas saben cuánto dinero tienen que pedir prestado y el tiempo que se requiere para su pago. Entonces preguntan a una institución prestataria cuál será el pago mensual con el interés establecido. También están al corriente de qué cantidad del pago mensual se destina al pago de los intereses, qué interés total han pagado desde que solicitaron el préstamo y cuánto adeudan al final de cada mes.

Escribir un programa en C que pueda ser usado por una institución que preste dinero para suministrar esta información a un cliente potencial. Suponer que se especifican la cantidad del préstamo, el interés anual y la duración del préstamo. La cantidad pagada mensualmente se calcula con:

$$A = i\,P\,(1 + i)^{n} / [(1 + i)^{n} - 1]$$

donde A = pago mensual, dólares
P = cantidad total del préstamo, dólares
i = interés mensual, expresado en decimal (esto es, 1/2 por ciento se debe escribir 0.005)
n = número total de pagos mensuales

El pago mensual de intereses se puede calcular con la fórmula

$$I = i\,B$$

donde I = pago mensual de intereses, dólares
B = principal actual, dólares

El principal actual es simplemente igual a la cantidad original del préstamo, menos la suma de los pagos previos destinados al capital principal. El pago mensual destinado al capital principal (esto es, la cantidad en la cual se reduce el préstamo) es simplemente

$$T = A - I$$

donde T = pago mensual destinado al capital principal.

Utilizar el programa para calcular el coste de una hipoteca de `50000` dólares a `25` años con un interés anual del `8` por `100`. Repetir los cálculos para un interés anual del `8.5` por `100`. ¿Qué impacto tiene el incremento del interés en el `0.5` por `100` en el pago total de la hipoteca?

d) El método utilizado para calcular el coste de una hipoteca usado en el problema anterior se conoce como método de *cuota fija*, ya que el pago mensual es el mismo. Supóngase que los pagos mensuales se calculan por el método del interés simple. Es decir, que la misma cantidad se aplica para reducir el préstamo cada mes. Por tanto

$$T = P / n$$

El interés mensual, sin embargo, dependerá de la cantidad del pago, esto es,

$$I = i\,B$$

Por tanto, el pago mensual total, $A = T + I$, decrecerá cada mes según disminuye el pago.

Escribir un programa en C para calcular el coste de una hipoteca utilizando este método de pago. Etiquetar claramente la salida. Usar el programa para calcular el coste de una hipoteca de `50000` dólares a `25` años con un interés anual del `8` por `100`. Comparar los resultados con los obtenidos en el apartado *c*).

e) Suponga que se dan una serie de puntos discretos (x_1, y_1), (x_2, y_2), ..., (x_n, y_n) leídos de la curva $y = f(x)$, donde x está acotada entre x_1 y x_n. Se desea aproximar el área encerrada por la curva descomponiendo la curva en un número de pequeños rectángulos y calculando el área de estos rectángulos. (Esto se conoce como la *regla del trapecio*.) La fórmula adecuada es

$$A = (y_1 + y_2)(x_2 - x_1)/2 + (y_2 + y_3)(x_3 - x_2)/2 + ... + (y_{n-1} + y_n)(x_n - x_{n-1})/2$$

Tenga en cuenta que la altura media de cada rectángulo viene dada por $(y_i + y_{i+1})/2$ y la anchura de cada rectángulo es igual a $(x_{i+1} - x_i)$; $i = 1, 2, ..., n - 1$.

Escribir un programa en C que implemente esta estrategia, usando una función para evaluar la fórmula matemática $y = f(x)$. Utilizar el programa para calcular el área bajo la curva $y = x^3$ entre los límites $x = 1$ y $x = 4$. Resolver este problema primero con `16` puntos espaciados, luego con `61` puntos y finalmente con `301` puntos. Observe que la precisión de la solución mejora según aumenta el número de puntos. (La respuesta exacta a este problema es `63.75`.)

f) El problema anterior describe un método conocido como la *regla del trapecio* para calcular el área encerrada por una curva $y = f(x)$, donde se usan una serie de valores tabulados (x_1, y_1), (x_2, y_2), ..., (x_n, y_n) para describir la curva. Si los valores tabulados de x están equiespaciados, entonces la ecuación dada en el problema anterior puede simplificarse a

$$A = (y_1 + 2y_2 + 2y_3 + 2y_4 + ... + 2y_{n-1} + y_n)\,h/2$$

donde h es la distancia entre dos valores sucesivos de x.

Otra técnica aplicable cuando hay un número par de intervalos igualmente espaciados (un número impar de puntos) es la *regla de Simpson*. La ecuación para implementar la regla de Simpson es

$$A = (y_1 + 4y_2 + 2y_3 + 4y_4 + 2y_5 + ... + 4y_{n-1} + y_n)\,h/3$$

Para un valor dado de h, este método proporciona mayor exactitud que la regla del trapecio. (Observe que el método requiere casi la misma complejidad computacional que la regla del trapecio.)

Escribir un programa en C para calcular el área encerrada por una curva usando ambas técnicas, suponiendo un número impar de puntos espaciados por igual. Implementar cada método con una función separada y utilizar otra función para evaluar $y(x)$.

Usar el programa para calcular el área encerrada por la curva

$$y = e^{-x}$$

donde x va de 0 a 1. Calcular el área usando cada método y comparar los resultados con la respuesta correcta de $A = 0.7468241$.

g) Otra técnica adicional para calcular el área encerrada por una curva es emplear el método de *Monte Carlo*, que hace uso de números generados aleatoriamente. Suponga que la curva $y = f(x)$ es positiva para cualquier valor de x entre los límites superior e inferior $x = a$ y $x = b$. Sea y^* el mayor valor de y dentro de estos límites. El método de Monte Carlo funciona así:

i) Empezar con un contador a cero.
ii) Generar un número aleatorio, r_x, de valor comprendido entre a y b.
iii) Evaluar $y(r_x)$.
iv) Generar un segundo número aleatorio, r_y, de valor comprendido entre 0 e y^*.
v) Comparar r_y con $y(r_x)$. Si r_y es menor o igual que $y(r_x)$, entonces este punto caerá sobre o debajo de la curva dada. En tal caso el contador se incrementa en 1.
vi) Repetir los pasos del ii) al v) un número grande de veces. Cada repetición se denomina un *ciclo*.
vii) Calcular la fracción F de puntos que caen en o bajo la curva tras completar un número de ciclos, dividiendo el valor del contador por el número total de ciclos. El área bajo la curva se obtiene como

$$A = Fy^* (b - a).$$

Escribir un programa en C que implemente esta estrategia. Usar este programa para calcular el área encerrada por la curva $y = e^{-x}$ entre los límites $a = 0$ y $b = 1$. Determinar cuántos ciclos son necesarios para obtener el resultado con una precisión de tres cifras significativas. Comparar el tiempo de cálculo requerido para este problema con el tiempo requerido para el problema anterior. ¿Qué método es mejor?

h) Una variable aleatoria normalmente distribuida, x, con media μ y desviación estándar σ, se puede generar con la fórmula

$$x = \mu + \sigma \frac{\sum_{i=1}^{N} r_i - N/2}{\sqrt{N/12}}$$

donde r_i es un número aleatorio uniformemente distribuido de valor comprendido entre 0 y 1. Normalmente se selecciona un valor $N = 12$ cuando se utiliza esta fórmula. La base de la fórmula es el *teorema del límite central*, que establece que un conjunto de valores medios de variables aleatorias uniformemente distribuidas tiende a estar uniformemente distribuido.

Escribir un programa en C que genere un número especificado de variables aleatorias normalmente distribuidas con una media y una desviación estándar dadas. Suponer que el número de variables aleatorias, la media y la desviación estándar son datos de entrada del programa. Generar cada variable aleatoria con una función que acepte la media y la desviación estándar como argumentos.

i) Escribir un programa en C que permita a una persona jugar a las tres en raya con la computadora. Escribir el programa de forma que la computadora pueda ser tanto el primero como el segundo jugador. Si la computadora es el primer jugador, el primer movimiento se generará al azar. Mostrar el estado del juego tras cada movimiento. La computadora debe saber cuándo gana cada jugador.

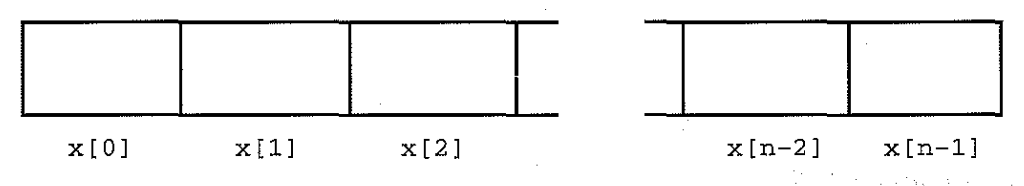
j) Escribir un programa completo en C que incluya una función recursiva para determinar el valor del *n*-ésimo número de Fibonacci, F_n, donde $F_n = F_{n-1} + F_{n-2}$ y $F_1 = F_2 = 1$ (ver Ejemplo 8.7). Considerar el valor de *n* como un dato de entrada.

CAPÍTULO 9

Arrays

Muchas aplicaciones requieren el procesamiento de múltiples datos que tienen características comunes (por ejemplo, un conjunto de datos numéricos, representados por x_1, x_2, ..., x_n). En tales situaciones es a menudo conveniente colocar los datos en un *array*, donde todos comparten el mismo nombre (por ejemplo x). Los datos individuales pueden ser caracteres, enteros, números en coma flotante, etc. Sin embargo, todos deben ser del mismo tipo de datos y con el mismo tipo de almacenamiento.

Cada elemento del array (esto es, cada dato individual) es referenciado mediante la especificación del nombre del array seguido por uno o más *índices*, con cada índice encerrado entre paréntesis cuadrados. Cada índice debe ser expresado como un entero no negativo. Así en un array de *n* elementos, los elementos del array son `x[0]`, `x[1]`, `x[2]`, ..., `x[n-1]`, como se ilustra en la Figura 9.1. El valor de cada índice puede ser expresado como una constante entera, una variable entera o una expresión entera más compleja.



x es un array unidimensional de *n* elementos

Figura 9.1.

El número de índices determinan la dimensionalidad del array. Por ejemplo, `x[i]` referencia a un elemento del array unidimensional `x`. Análogamente, `y[i][j]` referencia a un elemento del array bidimensional `y`. (Se puede pensar que un array bidimensional es una tabla, donde `y[i][j]` es el elemento `j` de la fila `i`.) Se pueden formular arrays de mayor dimensión, añadiendo índices adicionales de la misma manera (por ejemplo, `z[i][j][k]`).

Recuérdese que ya hemos utilizado arrays unidimensionales antes en este libro, conjuntamente con el procesamiento de cadenas de caracteres y líneas de texto. Por consiguiente, los arrays no son algo completamente nuevo, aunque nuestras referencias anteriores fueron casuales. Ahora consideraremos los arrays con mayor detalle. En particular se tratará la manera de definir y procesar arrays, el paso de arrays a funciones y la utilización de formaciones multidimensionales. Se considerarán tanto arrays numéricos como de caracteres. Inicialmente nos cen-

traremos en los arrays unidimensionales, si bien los arrays multidimensionales se considerarán en la sección 9.4.

9.1. DEFINICIÓN DE UN ARRAY

Los arrays se definen como las variables ordinarias, acompasando a cada nombre de array con una especificación de tamaño (número de elementos). Para un array unidimensional, el tamaño se especifica con una expresión entera positiva encerrada entre paréntesis cuadrados. En términos generales, una definición de array unidimensional puede expresarse como

```
tipo-de-almacenamiento tipo-dato array[expresión];
```

donde *tipo-de-almacenamiento* se refiere al tipo de almacenamiento del array, *tipo-dato* es el tipo de datos, *array* es el nombre del array y *expresión* es una expresión entera positiva que indica el número de elementos del array. El *tipo-de-almacenamiento* es opcional; los valores por omisión son `automatic` para arrays definidos dentro de una función o bloque, y `extern` para arrays definidos fuera de una función.

EJEMPLO 9.1. A continuación se muestran algunas definiciones típicas de arrays unidimensionales.

```
int x[100];

char texto[80];

static char mensaje[25];

static float n[12];
```

La primera línea establece que `x` es un array de `100` elementos enteros, y la segunda define `texto` como un array de `80` caracteres. En la tercera línea se define `mensaje` como un array estático de `25` caracteres, mientras que la cuarta establece que `n` es un array estático de `12` elementos en coma flotante.

Algunas veces es conveniente definir el tamaño de un array en términos de una constante simbólica en vez de una cantidad entera fija. Esto hace más fácil modificar un programa que utiliza un array, ya que todas las referencias al tamaño máximo del array (por ejemplo, en bucles `for` o en definiciones de arrays) pueden ser alteradas cambiando simplemente el valor de la constante simbólica.

EJEMPLO 9.2. Conversión de texto en minúsculas a mayúsculas. A continuación presentamos un programa completo que lee un array unidimensional de caracteres, convierte todos sus elementos a mayúsculas y escribe el array modificado. Programas similares se muestran en los Ejemplos 4.4, 6.9, 6.12 y 6.16.

```
/* leer una línea de texto en minúsculas y escribirla en mayúsculas */

#include <stdio.h>
#include <ctype.h>

#define TAMANO 80

main()
{
  char letras[TAMANO];
  int cont;

  /* leer la línea */

  for (cont = 0; cont < TAMANO; ++cont)
      letras[cont] = getchar();

  /* escribir la línea en mayúsculas */

  for (cont = 0; cont < TAMANO; ++cont)
      putchar(toupper(letras[cont]));
}
```

Observe que la constante simbólica `TAMANO` tiene asignado un valor de `80`. En la definición del array y en las dos instrucciones `for` aparece esta constante simbólica en vez de su valor. (Recuerde que *el valor de la constante simbólica será sustituido por la constante durante el proceso de compilación.*) Por tanto, para alterar el programa acomodándolo a diferentes tamaños del array, sólo debe cambiarse la instrucción `#define`.

Por ejemplo, para alterar el programa anterior de modo que procese un array de `60` elementos, simplemente se reemplaza la instrucción `#define` original por

```
#define TAMANO 60
```

Este único cambio se encarga de todas las alteraciones necesarias del programa; no hay posibilidad de pasar por alto algunas modificaciones necesarias.

Los arrays automáticos, a diferencia de las variables automáticas, no pueden ser inicializados. Sin embargo, las definiciones de arrays externos y estáticos pueden incluir, si se desea, la asignación de valores iniciales. Los valores iniciales deben aparecer en el orden en que serán asignados a los elementos individuales del array, encerrados entre llaves y separados por comas. La forma general es

```
tipo-de-almacenamiento tipo-dato array[expresión] =
                        {valor 1, valor 2, . . ., valor n};
```

donde `valor 1` se refiere al valor del primer elemento del array, `valor 2` al valor del segundo elemento, y así sucesivamente. La presencia de la `expresión`, que indica el número de elementos del array, es opcional cuando están presentes los valores iniciales.

EJEMPLO 9.3. A continuación se muestran varias definiciones de arrays que incluyen la asignación de valores iniciales.

```
int digitos[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

static float x[6] = {0, 0.25, 0, -0.50, 0, 0};

char color[4] = {'R', 'O', 'J', 'O'};
```

Observe que `x` es un array estático. Los otros dos arrays (`digitos` y `color`) se supone que son externos en virtud de su colocación dentro del programa.

El resultado de estas asignaciones iniciales, en términos de los elementos individuales del array, es el siguiente. (Recordar que los índices de un array de `n` elementos van de `0` a `n-1`.)

```
digitos[0] = 1      x[0] =  0       color[0] = 'R'
digitos[1] = 2      x[1] =  0.25    color[1] = 'O'
digitos[2] = 3      x[2] =  0       color[2] = 'J'
digitos[3] = 4      x[3] = -0.50    color[3] = 'O'
digitos[4] = 5      x[4] =  0
digitos[5] = 6      x[5] =  0
digitos[6] = 7
digitos[7] = 8
digitos[8] = 9
digitos[9] = 10
```

Todos los elementos del array que no tienen asignados valores iniciales explícitos serán puestos automáticamente a cero. Esto incluye al resto de los elementos de un array en el que un cierto número de elementos tienen asignados un valor distinto de cero.

EJEMPLO 9.4. Considerar las siguientes definiciones de arrays.

```
int digitos[10] = {3, 3, 3};

static float x[6] = {-0.3, 0, 0.25};
```

Los resultados, elemento a elemento, son:

```
digitos[0] = 3      x[0] = -0.3
digitos[1] = 3      x[1] =  0
digitos[2] = 3      x[2] =  0.25
digitos[3] = 0      x[3] =  0
digitos[4] = 0      x[4] =  0
digitos[5] = 0      x[5] =  0
digitos[6] = 0
digitos[7] = 0
digitos[8] = 0
digitos[9] = 0
```

En cada caso, todos los elementos del array se ponen automáticamente a cero, excepto los que han sido explícitamente inicializados dentro de la definición del array. Observe que los valores repetidos (por ejemplo `3, 3, 3`) se deben poner separadamente.

El tamaño del array no necesita ser especificado explícitamente cuando se incluyen los valores iniciales como una parte de la definición del array. Con un array numérico, el tamaño del array será fijado automáticamente igual que el número de valores incluidos dentro de la definición.

EJEMPLO 9.5. Considerar las siguientes definiciones de arrays, que son variaciones de las definiciones mostradas en los Ejemplos 9.3 y 9.4.

```
int digitos[] = {1, 2, 3, 4, 5, 6};

static float x[] = {0, 0.25, 0, -0.5};
```

Así `digitos` será un array de seis elementos y x un array estático de cuatro elementos en coma flotante. Los elementos individuales tendrán asignados los siguientes valores. (*Observe los corchetes vacíos en las declaraciones de los arrays.*)

```
digitos[0] = 1        x[0] =  0
digitos[1] = 2        x[1] =  0.25
digitos[2] = 3        x[2] =  0
digitos[3] = 4        x[3] = -0.5
digitos[4] = 5
digitos[5] = 6
```

Las cadenas de caracteres (arrays de caracteres) son manejadas de modo algo diferente, como se indicó en la sección 2.6. En particular, cuando se le asigna una cadena de caracteres constante a un array estático o externo como parte de la definición, la especificación del tamaño del array normalmente se omite. El tamaño adecuado será asignado automáticamente. Esto incluirá la provisión para el carácter nulo `\0`, que se añade automáticamente al final de cada cadena de caracteres (ver Ejemplo 2.26).

EJEMPLO 9.6. Considerar las dos definiciones siguientes de arrays de caracteres. Cada una de ellas incluye la asignación inicial de la constante de cadena de caracteres `"ROJO"`. Sin embargo, el primer array se define como un array de cuatro elementos, mientras que no se especifica el tamaño del segundo array.

```
char color[4] = "ROJO";

char color[] = "ROJO";
```

El resultado de estas asignaciones iniciales no es el mismo debido al carácter nulo, `\0`, que se añade automáticamente al final de la segunda cadena. De este modo, los elementos del primer array son

```
color[0] = 'R'
color[1] = 'O'
color[2] = 'J'
color[3] = 'O'
```

mientras que los elementos del segundo array son

```
color[0] = 'R'
color[1] = 'O'
color[2] = 'J'
color[3] = 'O'
color[4] = '\0'
```

Por tanto, la primera forma es incorrecta, ya que el carácter nulo `\0` no se incluyó en el array.

La definición del array podría haberse escrito como

```
char color[5] = "ROJO";
```

Esta definición es correcta, ya que ahora definimos un array de cinco elementos que incluye un elemento para el carácter nulo. Sin embargo, muchos programadores prefieren la primera forma, que omite el especificador de tamaño.

Si el programa requiere una *declaración* de un array unidimensional (porque el array está definido en cualquier otra parte del programa), la declaración se escribe de la misma manera que la definición del array con las siguientes excepciones:

1. Los corchetes pueden estar vacíos, ya que el tamaño ha sido especificado como parte de la definición. Las declaraciones de arrays se escriben habitualmente de esta manera.
2. No se pueden incluir valores iniciales en la declaración.

Estas reglas se aplican tanto a declaraciones de argumentos formales dentro de las funciones como a declaraciones de variables externas. Sin embargo, las reglas para definir argumentos formales *multidimensionales* son más complejas (ver sección 9.4).

EJEMPLO 9.7. A continuación se muestra el esqueleto de la estructura de un programa en C que hace uso de arrays externos.

Primer archivo

```
int c[] = {1, 2, 3};            /* DEFINICION de array externo */

char mensaje[] = "¡Hola!";      /* DEFINICION de array externo */

extern void func1(void);        /* prototipo de función */

main()
{
  . . . . .
}
```

Segundo archivo

```
extern int c[];                 /* DECLARACION de array externo */

extern char mensaje[];          /* DECLARACION de array externo */
```

```
extern void func1(void)        /* definición de función */
{
   . . . . .
}
```

Este esquema de programa incluye dos arrays externos, `c` y `mensaje`. El primer array (`c`) es un array de tres elementos enteros que se definen e inicializan en el primer archivo. El segundo array (`mensaje`) es un array de caracteres definido e inicializado también en el primer archivo. Los arrays están luego *declarados* en el segundo archivo, ya que son arrays globales que deben ser reconocidos en todo el programa.

Ni las definiciones de arrays en el primer archivo ni las declaraciones de arrays en el segundo archivo incluyen especificación explícita de tamaño. Estas especificaciones se permiten en el primer archivo, pero se omiten debido a la inicialización. Además, las especificaciones de tamaño de los arrays no tienen sentido en el segundo archivo porque el tamaño de los arrays ya ha sido fijado.

9.2. PROCESAMIENTO DE UN ARRAY

En C no se permiten operaciones que involucren arrays completos. Así, si `a` y `b` son arrays similares (mismo tipo de datos, misma dimensionalidad y mismo tamaño), operaciones de asignación, de comparación, etc., deben realizarse elemento por elemento. Esto se hace normalmente dentro de un bucle, donde cada pasada por el bucle se usa para procesar un elemento del array. El número de pasadas por el bucle será por tanto igual al número de elementos del array a procesar.

Ya hemos visto varios ejemplos donde se procesan los elementos individuales de un array de caracteres de una u otra manera (ver Ejemplos 4.4, 4.19, 6.9, 6.12, 6.16, 6.19, 6.20, 6.32, 6.34, 8.3, 8.5 y 9.2). Los arrays numéricos se procesan de la misma manera. En un array numérico, cada elemento del array representa una cantidad numérica, como se ilustra en el ejemplo siguiente.

EJEMPLO 9.8. Desviaciones respecto a la media. Supongamos que queremos leer una lista de n cantidades en coma flotante y luego calcular su media, como en el Ejemplo 6.17. Sin embargo, además de calcular simplemente la suma, se calculará también la *desviación* de cada cantidad numérica respecto de la media, utilizando la fórmula

$$d = x_i - media$$

donde x_i representa cada una de las cantidades dadas, $i = 1, 2, ..., n$, y *media* la media calculada.

Para resolver este problema debemos almacenar las cantidades dadas en un array unidimensional de elementos en coma flotante. Ésta es la parte esencial del programa. La razón, que debe quedar clara, es la que sigue.

En todos los ejemplos anteriores donde se ha calculado la media de una lista de números, cada número era sustituido por su sucesor en la lista (ver Ejemplos 6.10, 6.13, 6.17 y 6.31). Por tanto, cada número individual no estaba disponible para cálculos posteriores una vez que el siguiente era introducido. Sin embargo, ahora estos números deben ser retenidos dentro de la computadora para calcular su correspondiente desviación respecto de la media determinada. Por tanto, los almacenamos en un array unidimensional llamado `lista`.

Definimos `lista` como un array de `100` elementos en coma flotante. Sin embargo, no necesitamos usar todos los elementos. En su lugar, deberemos introducir en la variable entera `n` una cantidad entera positiva (que no pase de `100`) para especificar el número de elementos reales.

A continuación se muestra el programa completo en C.

```
/* calcular la media de n números, después calcular la desviación de
   cada número respecto a la media */

#include <stdio.h>

main()
{
  int n, cont;
  float media, d, suma = 0;
  float lista[100];

  /* leer el valor de n */
  printf("\n¿Cuantos números para calcular la media? ");
  scanf("%d", &n);
  printf("\n");

  /* leer los números y calcular su suma */
  for (cont = 0; cont < n; ++cont) {
      printf("i = %d   x = ", cont + 1);
      scanf("%f", &lista[cont]);
      suma += lista[cont];
  }

  /* calcular la media y escribir la respuesta */
  media = suma / n;
  printf("\nLa media es %5.2f\n\n", media);

  /* calcular y escribir las desviaciones respecto de la media */
  for (cont = 0; cont <n; ++cont) {
      d = lista[cont] - media;
     printf("i = %d  x = %5.2f  d = %5.2f\n", cont + 1, lista[cont], d);
  }
}
```

Observe que la segunda función `scanf` (dentro del primer bucle `for`) incluye un ampersand (`&`) delante de `lista[cont]`, ya que refiere a un elemento simple del array y no al array entero (ver sección 4.4).

Ahora supongamos que el programa se ejecuta usando las siguientes cinco cantidades numéricas: $x_1 = 3$, $x_2 = -2$, $x_3 = 12$, $x_4 = 4.4$, $x_5 = 3.5$. La sesión interactiva, incluyendo la entrada de datos y los resultados calculados, se muestra a continuación. Las respuestas del usuario están subrayadas.

```
¿Cuantos números para calcular la media? 5

i = 1   x = 3
i = 2   x = -2
i = 3   x = 12
i = 4   x = 4.4
i = 5   x = 3.5
```

```
La media es  4.18

i = 1    x =  3.00    d = -1.18
i = 2    x = -2.00    d = -6.18
i = 3    x = 12.00    d =  7.82
i = 4    x =  4.40    d =  0.22
i = 5    x =  3.50    d = -0.68
```

En algunas aplicaciones puede desearse asignar valores iniciales a los elementos de un array. Esto requiere que el array sea definido global o localmente (dentro de una función) como un array estático. El siguiente ejemplo ilustra el uso de un array global de formación.

EJEMPLO 9.9. Desviaciones respecto a la media (revisión). Calculemos de nuevo la media de un conjunto dado de números y luego computemos la desviación de cada número respecto a la media, como en el Ejemplo 9.8. Sin embargo, ahora asignaremos los números dados al array dentro de su definición. Para hacerlo, movemos la definición del array `lista` fuera de `main`. Así `lista` será un array externo. Además quitaremos la especificación de tamaño explícita de la definición, ya que el número de valores iniciales determinará el tamaño del array.

Los valores iniciales incluidos en el siguiente programa son los mismos cinco valores que fueron usados como datos en el Ejemplo 9.8. Para ser consistentes asignaremos un valor inicial a `n`. Esto se puede realizar definiendo `n` bien como una variable automática dentro de `main` o bien como una variable externa. Se ha elegido esto último para que estén agrupadas las asignaciones iniciales, que en otro caso podrían ser introducidas como datos de entrada.

A continuación se muestra el programa completo.

```
/* calcular la media de n números, después calcular la desviación
   de cada número respecto a la media */

#include <stdio.h>

int n = 5;
float lista[] = {3, -2, 12, 4.4, 3.5};

main()

{
   int cont;
   float media, d, suma = 0;

   /* calcular la media y escribir la respuesta */
   for (cont = 0; cont < n; ++cont)
       suma += lista[cont];
   media = suma / n;
   printf("\nLa media es %5.2f\n\n", media);

   /* calcular y escribir las desviaciones respecto de la media */
   for (cont = 0; cont <n; ++cont) {
       d = lista[cont] - media;
      printf("i = %d   x = %5.2f   d = %5.2f\n", cont + 1, lista[cont], d);
   }
}
```

Observe que esta versión del programa no requiere ningún dato de entrada.

La ejecución del programa genera la siguiente salida:

```
La media es  4.18

i = 1    x =  3.00   d = -1.18
i = 2    x = -2.00   d = -6.18
i = 3    x = 12.00   d =  7.82
i = 4    x =  4.40   d =  0.22
i = 5    x =  3.50   d = -0.68
```

9.3. PASO DE ARRAYS A FUNCIONES

Un array completo se puede pasar a una función como argumento. Sin embargo, la manera en la que el array se pasa difiere mucho de la de una variable ordinaria.

Para pasar un array a una función, el nombre del array debe aparecer solo, sin corchetes ni índices, como un argumento real en la llamada a la función. El correspondiente argumento formal se escribe de la misma manera, pero debe ser declarado como un array en la declaración de los argumentos formales. Cuando se declara un array unidimensional como un argumento formal, el array se escribe con un par de corchetes vacíos. El tamaño del array no se especifica en la declaración de argumentos formales.

Se debe tener cuidado al escribir prototipos de funciones que incluyan argumentos de array. Una pareja vacía de corchetes debe seguir al nombre de cada argumento de array, indicando de este modo que el argumento es un array. Si no se incluyen los nombres de los argumentos en una declaración de función, entonces una pareja vacía de corchetes debe seguir al tipo de datos de argumento de array.

EJEMPLO 9.10. El siguiente esquema de programa ilustra el paso de un array desde la parte principal del programa a una función.

```
float media(int a, float x[]);    /* prototipo de función */

main()
{
   int n;                         /* DECLARACION de variable */
   float med;                     /* DECLARACION de variable */
   float lista[100];              /* DEFINICION de array */

   . . . . .

   med = media(n, lista);

   . . . . .
}
float media(int a, float x[])     /* DEFINICION de función */
{
  . . . . .
}
```

Dentro de `main` vemos una llamada a la función `media`. Esta llamada a función contiene dos argumentos reales, la variable entera `n` y el array unidimensional en coma flotante `lista`. Observe que `lista` aparece como una variable ordinaria en la llamada a la función; es decir, no se incluyen los corchetes.

La primera línea de la definición de la función incluye dos argumentos formales, `a` y `x`. La declaración de argumentos formales establece que `a` es una variable entera y `x` un array unidimensional en coma flotante. Existe pues una correspondencia entre el argumento real `n` y el argumento formal `a`. Análogamente, existe una correspondencia entre el argumento real `lista` y el argumento formal `x`. Observe que el tamaño de `x` no se especifica dentro de la declaración formal de argumentos.

Observe que el prototipo de función se podría haber escrito sin los nombres de los argumentos, como

```
float media(int, float[]);     /* prototipo de función */
```

Cualquiera de las dos formas es válida.

Ya hemos discutido el hecho de que los argumentos se pasan a una función por valor cuando los argumentos son variables ordinarias (ver sección 7.5). Sin embargo, cuando se pasa un array a una función, *no* se pasan a la función los valores de los elementos del array. En su lugar, el nombre del array se interpreta como la *dirección* del primer elemento del array (la dirección de la posición de memoria que contiene el primer elemento del array). Esta dirección se asigna al correspondiente argumento formal cuando se llama a la función. El argumento formal se convierte por tanto en un *puntero* al primer elemento del array (más sobre esto en el próximo capítulo). Los argumentos pasados de esta manera se dice que son pasados *por referencia* en vez de por valor.

Cuando se hace una referencia a un elemento del array dentro de la función, el valor del índice del elemento se añade al valor del puntero para indicar la dirección del elemento especificado. Por tanto, cualquier elemento del array puede ser accedido desde dentro de la función. Además, *si un elemento del array es modificado dentro de la función, esta modificación será reconocida en la parte del programa desde la que se hizo la llamada* (en realidad, en todo el ámbito del array).

EJEMPLO 9.11. A continuación se muestra un programa sencillo en C que pasa un array de tres elementos enteros a una función donde se modifican dichos elementos. Los valores de los elementos del array se escriben en tres partes del programa para ilustrar los efectos de las modificaciones.

```
#include <stdio.h>

void modificar(int a[]);     /* prototipo de función */

main()
{
  int cont, a[3];             /* definición de array */

  printf("\nDesde main, antes de llamar a la función:\n");
  for (cont = 0; cont <= 2; ++cont) {
      a[cont] = cont + 1;
      printf("a[%d] = %d\n", cont, a[cont]);
  }
```

```
    modificar(a);

    printf("\nDesde main, después de llamar a la función:\n");
    for (cont = 0; cont <= 2; ++cont)
        printf("a[%d] = %d\n", cont, a[cont]);
}

void modificar(int a[])     /* definición de función */
{
    int cont;

    printf("\nDesde la función, después de modificar los valores:\n");
    for (cont = 0; cont <= 2; ++cont) {
        a[cont] = -9;
        printf("a[%d] = %d\n", cont, a[cont]);
    }
    return;
}
```

A los elementos del array se les asignan los valores `a[0] = 1`, `a[1] = 2` y `a[2] = 3` en el primer bucle de `main`. Estos valores se escriben tras ser asignados. Entonces el array se pasa a la función `modificar`, donde se le asigna el valor `-9` a cada elemento del array. Estos nuevos valores se escriben desde dentro de la función. Finalmente, los valores del array se escriben de nuevo en `main`, una vez que el control haya sido transferido desde `modificar` hasta `main`.

Cuando el programa se ejecuta, se genera la siguiente salida:

```
Desde main, antes de llamar a la función:
a[0] = 1
a[1] = 2
a[2] = 3

Desde la función, después de modificar los valores:
a[0] = -9
a[1] = -9
a[2] = -9

Desde main, después de llamar a la función:
a[0] = -9
a[1] = -9
a[2] = -9
```

Estos resultados muestran que los elementos de `a` se modifican dentro de `main` como resultado de los cambios realizados dentro de `modificar`.

EJEMPLO 9.12. Consideremos ahora una variación del programa anterior. El programa actual incluye la utilización de variables globales y la transferencia tanto de una variable local como de un array a la función.

```
#include <stdio.h>

int a = 1;                          /* variable global */
void modificar(int b, int c[]);     /* prototipo de función */
```

```
main()
{
   int b = 2;              /* variable local */
   int cont , c[3];        /* definición de array */

   printf("\nDesde main, antes de llamar a la función:\n");
   printf("a = %d   b = %d\n", a, b);
   for (cont = 0; cont <= 2; ++cont) {
        c[cont] = 10 * (cont + 1);
        printf("c[%d] = %d\n", cont, c[cont]);
   }

   modificar(b, c);           /* acceso a la función */
   printf("\nDesde main, después de llamar a la función:\n");
   printf("a = %d   b = %d\n", a, b);
   for (cont = 0; cont <= 2; ++cont)
        printf("c[%d] = %d\n", cont, c[cont]);
}

void modificar(int b , int c[])        /* definición de función */
{
   int cont;

   printf("\nDesde la función, después de modificar los valores:\n");
   a = -999;
   b = -999;
   printf("a = %d   b = %d\n", a, b);
   for (cont = 0; cont <= 2; ++cont) {
        c[cont] = -9;
        printf("c[%d] = %d\n", cont, c[cont]);
   }
   return;
}
```

Cuando se ejecuta el programa, se genera la siguiente salida:

```
Desde main, antes de llamar a la función:
a = 1   b = 2
c[0] = 10
c[1] = 20
c[2] = 30

Desde la función, después de modificar los valores:
a = -999   b = -999
c[0] = -9
c[1] = -9
c[2] = -9

Desde main, después de llamar a la función:
a = -999   b = 2
c[0] = -9
c[1] = -9
c[2] = -9
```

Vemos ahora que el valor de `a` y los elementos de `c` son modificados dentro de `main` como resultado de los cambios realizados en `modificar`. Sin embargo, el cambio realizado a `b` se confina a la función, como era de esperar. (Comparar estos resultados con los obtenidos en el último ejemplo y en el Ejemplo 7.12.)

El hecho de que un array pueda ser modificado globalmente dentro de una función proporciona un mecanismo adecuado para mover múltiples datos a o desde una función a la parte del programa desde la que se hizo la llamada. Simplemente se pasa el array a la función y se alteran sus elementos dentro de ella. O, si el array original debe ser preservado, se copia el array (elemento por elemento) dentro de la parte del programa que hace la llamada, se pasa la copia a la función y se realizan las modificaciones. El programador debe tener cierto cuidado al modificar un array dentro de una función, ya que es muy fácil modificarlo de modo no intencionado fuera de la función.

EJEMPLO 9.13. Reordenación de una lista de números. Consideremos el famoso problema de reordenar una lista de `n` enteros en una secuencia de valores algebraicos crecientes. Escribamos un programa que realice la reordenación de modo que no se use almacenamiento innecesario. Por tanto, el programa contendrá sólo un array: un array unidimensional de enteros llamado `x`, en el que se reordenarán los elementos de uno en uno.

La reordenación empezará recorriendo todo el array para encontrar el menor de los números. Este número será intercambiado con el primer elemento del array, colocando así el menor número al principio de la lista. El resto de los `n-1` elementos del array se recorrerán para encontrar el menor, que se intercambiará con el segundo número. El resto de los `n-2` números se recorrerán buscando el menor, que se intercambia con el tercer número, y así sucesivamente, hasta que se haya reordenado el array completo. La reordenación completa requerirá un total de `n-1` pasadas por el array, pero cada vez el trozo del array a considerar es menor.

Para encontrar el menor de los números en cada pasada, comparamos secuencialmente cada número del array, `x[i]`, con el número de comienzo, `x[elem]`, donde `elem` es una variable entera que se usa para identificar un elemento particular del array. Si `x[i]` es menor que `x[elem]`, entonces se intercambian los dos números; en otro caso se dejan los dos números en sus posiciones originales. Una vez que este procedimiento se ha aplicado a todo el array, el primer número del array será el menor. Después se repite este proceso `n-2` veces, para un total de `n-1` pasadas (`elem=0,1,...,n-2`).

La única cuestión que queda es cómo se intercambian realmente los números. Para realizar el intercambio, primero almacenamos temporalmente el valor de `x[elem]` para referenciarlo en el futuro. Después asignamos el valor actual de `x[i]` a `x[elem]`. Finalmente asignamos el valor *original* de `x[elem]`, que fue temporalmente almacenado, a `x[i]`. El intercambio está ahora completo.

La estrategia descrita anteriormente puede ser escrita en C como sigue:

```
/* reordenar todos los elementos del array */
for (elem = 0; elem < n - 1; ++elem)
    /* encontrar el menor del resto de los elementos */
    for (i = elem + 1; i < n; ++i)
         if (x[i] < x[elem]) {
            /* intercambiar los dos elementos */
            temp = x[elem];
            x[elem] = x[i];
            x[i] = temp;
         }
```

Estamos suponiendo que `elem` e `i` son variables enteras que se usan como contadores y que `temp` es una variable entera que se usa para almacenar temporalmente el valor de `x[elem]`.

Ahora simplemente queda añadir las definiciones necesarias de variables y del array, y las instrucciones de entrada/salida. A continuación se muestra el programa completo en C.

```
/* reordenar un array unidimensional de enteros, de menor a
   mayor */

#include <stdio.h>

#define TAM 100

void reordenar(int n, int x[]);

main()
{
  int i, n, x[TAM];

  /* leer el valor de n */
  printf("\n¿Cuantos números serán introducidos? ");
  scanf("%d", &n);
  printf("\n");

  /* leer la lista de números */

  for (i = 0; i < n; ++i) {
      printf("i = %d   x = ", i + 1);
      scanf("%d", &x[i]);
  }
  /* reordenar todos los elementos del array */
  reordenar(n, x);

  /* escribir la lista reordenada de números */
  printf("\n\nLista de números reordenada:\n\n");
  for (i = 0; i < n; ++i)
      printf("i = %d   x = %d\n", i + 1, x[i]);
}

void reordenar(int n, int x[])   /* reordenar la lista de números */

{
  int i, elem, temp;

  for (elem = 0; elem < n - 1; ++elem)
      /* encontrar el menor del resto de los elementos */
      for (i = elem + 1; i < n; ++i)
          if (x[i] < x[elem]) {
              /* intercambiar los dos elementos */
              temp = x[elem];
              x[elem] = x[i];
              x[i] = temp;
          }
  return;
}
```

En este programa x se define inicialmente como un array de 100 elementos enteros. (Observe el uso de la constante simbólica TAM para definir el tamaño de x.) Primero se lee un valor para n, seguido por los valores numéricos de los primeros n elementos de x (x[0], x[1], ..., x[n-1]). Tras la entrada de datos, n y x se pasan a la función reordenar, donde los primeros n elementos de x se ordenan de modo ascendente. Al final del programa, los elementos reordenados de x se escriben en main.

La declaración de reordenar que aparece en main está escrita como un prototipo de función, como ejemplo de buena práctica de programación. Observe la manera en que se escriben los argumentos de la función. En particular, observe que el segundo argumento se identifica como un array entero por los corchetes que siguen al nombre del array, esto es, int x[]. Los corchetes son necesarios como una parte de la especificación de este argumento.

Supongamos ahora que el programa se utiliza para reordenar los siguientes seis números: 595 78 -1505 891 -29 -7. El programa generará el siguiente diálogo interactivo. (Como siempre, las respuestas del usuario están subrayadas.)

```
¿Cuantos números serán introducidos? 6

i = 1    x = 595
i = 2    x = 78
i = 3    x = -1505
i = 4    x = 891
i = 5    x = -29
i = 6    x = -7

Lista de números reordenada:

i = 1    x = -1505
i = 2    x = -29
i = 3    x = -7
i = 4    x = 78
i = 5    x = 595
i = 6    x = 891
```

Debe indicarse que *la instrucción* return *no puede ser usada para devolver un array*, ya que return sólo puede devolver expresiones *unievaluadas* a la parte del programa que hizo la llamada. Por tanto, si se tienen que pasar los elementos de un array a la parte del programa que hace la llamada, el array debe estar definido bien como un array externo cuyo ámbito incluya tanto la función como la parte del programa desde la que se llama a la función, o bien debe ser pasado a la función como un argumento formal.

EJEMPLO 9.14. Un generador de «pig latin». «Pig latin» es una forma codificada de escribir y de hablar que suelen usar los niños como juego. Una palabra en «pig latin» se forma transponiendo el primer sonido (generalmente la primera letra) de una palabra original al final de la misma y añadiéndole luego la letra «a». De este modo, la palabra «perro» se convierte en «erropa», «computadora» en «omputadoraca», «piglatin» en «iglatinpa» (o «igpa atinla» si se considera como dos palabras separadas), y así sucesivamente.

Escribamos un programa en C que acepte una línea de texto y escriba su correspondiente texto en «pig latin». Supondremos que cada mensaje textual puede estar contenido en una línea de 80 columnas, con un solo espacio en blanco entre cada dos palabras sucesivas. (Realmente, se exigirá que el mensaje en «pig latin» no exceda de 80 caracteres. Por tanto, el mensaje original debe constar de algo menos de 80 caracteres, ya que el correspondiente mensaje en «pig latin» tendrá mayor longitud por el añadido de la letra

«a» al final de cada palabra.) Por simplicidad, sólo transpondremos la primera letra (no el primer sonido) de cada palabra. Además ignoraremos cualquier consideración especial que pudiera darse a las letras mayúsculas y a los signos de puntuación.

Utilizaremos dos arrays en este programa. Un array contendrá la línea de texto original y el otro el «pig latin» correspondiente.

La estrategia general de operación contendrá los siguientes pasos principales:

1. Inicializar ambos arrays, asignando espacios en blanco a todos los elementos.
2. Leer una línea entera de texto (varias palabras).
3. Determinar el número de palabras de la línea (contando el número de espacios en blanco que van seguidos por un carácter no blanco).
4. Convertir las palabras a «pig latin», palabra a palabra, de la siguiente manera:
   a) Localizar el final de la palabra.
   b) Pasar la primera letra al final de la palabra y luego añadir una «a».
   c) Localizar el principio de la siguiente palabra.
5. Escribir la línea de «pig latin» completa.

Se continuará repitiendo este procedimiento, hasta que la computadora lea una línea de texto cuyas tres primeras letras sean «fin» (o «FIN»).

Para implementar esta estrategia haremos uso de dos indicadores, llamados `m1` y `m2`, respectivamente. El primer indicador (`m1`) señalará la posición del comienzo de una palabra en particular dentro de la línea de texto original. El segundo indicador (`m2`) señalará el final de la palabra.Observe que el carácter en la columna anterior a la indicada por `m1` será un espacio en blanco (excepto en la primera palabra). También será un espacio en blanco el carácter en la columna posterior a la señalada por `m2`.

Este programa se estructura en variar funciones para realizar cada una de las tareas principales. Sin embargo, antes de discutir las funciones individuales definiremos las siguientes variables de programa:

`texto` = array unidimensional de caracteres que representa la línea de texto original
`piglatin` = array unidimensional de caracteres que representa la nueva línea de texto (en «pig latin»)
`palabras` = variable de tipo entero que indica el número de palabras existentes en cada línea de texto dada
`n` = variable entera que se usa como contador de palabras (`n=1,2,...,palabras`)
`cont` = variable entera que se usa como contador de caracteres dentro de cada línea de texto (`cont=0,1,2,...,79`)

Haremos uso también de las variables enteras `m1` y `m2` discutidas anteriormente.

Volvamos ahora al esquema general del programa presentado anteriormente. El primer paso, la inicialización de los arrays, se puede realizar directamente con la siguiente función:

```
/* inicializar el array de caracteres con espacios en blanco */

void inicializar(char texto[], char piglatin[])

{
  int cont;

  for (cont = 0; cont < 80; ++cont)
      texto[cont] = piglatin[cont] = ' ';
  return;
}
```

El paso 2 puede realizarse con una función sencilla. Este procedimiento contendrá un bucle `while` que continuará leyendo caracteres desde el teclado hasta que se detecte el fin de línea. Esta secuencia de caracteres formará los elementos del array de caracteres `texto`. A continuación se muestra la función completa.

```
/* leer una línea de texto */

void leerentrada(char texto[])

{
   int cont = 0;
   char c;

   while ((c = getchar()) != '\n') {
         texto[cont] = c;
         ++cont;
   }
   return;
}
```

El paso 3 del esquema general es igualmente directo. Simplemente buscamos en la línea origina espacios en blanco seguidos de caracteres distintos de blanco. El contador de palabras (`palabras`) se incrementa cada vez que se encuentra un solo espacio en blanco. A continuación se muestra la rutina de contar palabras.

```
/* examinar la línea de texto y contar el número de palabras */

int contarpalabras(char texto[])

{
   int cont, palabras = 1;

   for (cont = 0; cont < 79; ++cont)
       if (texto[cont] == ' ' && texto[cont + 1] != ' ')
           ++palabras;
   return(palabras);
}
```

Ahora consideraremos el paso 4 (convertir el texto en «pig latin»), que es realmente el núcleo del programa. La lógica para realizar esto es algo más compleja, ya que necesita tres operaciones individuales pero a su vez relacionadas. Primero debemos identificar el final de cada palabra encontrando el primer espacio en blanco detrás de `m1`. Después asignamos los caracteres que componen la palabra al array de caracteres `piglatin`, con el primer carácter al final de la palabra. Finalmente debemos añadir la letra «a» y restablecer el indicador inicial para que identifique el principio de la siguiente palabra.

La lógica debe ser tratada cuidadosamente, porque la nueva línea de texto será más larga que la línea original (debido a la «a» añadida al final). Por esto, los caracteres en la primera palabra «pig latin» ocuparán las posiciones desde `m1` hasta `m2+1`. Los caracteres de la segunda palabra ocuparán las posiciones desde `m1+1` hasta `m2+2` (observe que éstos son los nuevos valores de `m1` y `m2`), y así sucesivamente. Estas reglas se pueden generalizar como sigue.

Primero, para la palabra número `n`, transferir todos los caracteres excepto el primero de la línea original a la nueva línea. Esto puede realizarse escribiendo

```
for (cont = m1; cont < m2; ++cont)
    piglatin[cont + (n - 1)] = texto[cont + 1];
```

Los dos últimos caracteres (el primer carácter de la palabra original más la letra «a») pueden añadirse de la siguiente manera:

```
piglatin[m2 + (n - 1)] = texto[m1];
piglatin[m2 + n] = 'a';
```

Después modificamos el valor de `m1`:

```
 m1 = m2 + 2;
```

para prepararlo para la siguiente palabra. Este grupo de cálculos se repite para cada palabra de la línea original.

A continuación se muestra la función que realiza todo esto.

```
/* convertir cada palabra en "pig-latin" */

void convertir(int palabras, char texto[], char piglatin[])

{
   int n, cont;
   int m1= 0;          /* indicador -> comienzo de la palabra */
   int m2;             /* indicador -> final de la palabra */

   /* convertir cada palabra */
   for (n = 1; n <= palabras; ++n) {

       /* localizar el final de la palabra actual */
       cont = m1;
       while (texto[cont] != ' ')
             m2 = cont++;

       /* transponer la primera letra y añadir una 'a' */
       for (cont = m1; cont < m2; ++cont)
           piglatin[cont + (n - 1)] = texto[cont + 1];
       piglatin[m2 + (n - 1)] = texto[m1];
       piglatin[m2 + n] = 'a';

       /* reinicializar el indicador inicial */
       m1 = m2 + 2;
   }
   return;
}
```

El paso 5 (escribir el «pig latin») requiere poco más que un bucle `for`. La función completa puede escribirse como

```
/* escribir la línea de texto en "pig-latin" */

void escribirsalida(char piglatin[])
{
```

```
    int cont = 0;

    for (cont = 0; cont < 80; ++cont)
        putchar(piglatin[cont]);
    printf("\n");
    return;
}
```

Consideremos ahora la parte principal del programa. Esto no es nada más que un grupo de definiciones y declaraciones, un mensaje inicial, un bucle `do - while` que permite la ejecución repetida del programa (hasta que la palabra «fin» se detecta, tanto en mayúsculas como en minúsculas, como primera palabra en una línea de texto) y un mensaje de finalización. El bucle `do - while` se puede hacer que continúe de forma indefinida usando la comprobación `(palabras >= 0)` al final del bucle. Como `palabras` tiene asignado un valor inicial de `1` y su valor no decrece, la comprobación será siempre verdadera.

El programa completo se muestra a continuación.

```
/* convertir un texto a "pig latin", línea a línea */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

void inicializar(char texto[], char piglatin[]);
void leerentrada(char texto[]);
int contarpalabras(char texto[]);
void convertir(int palabras, char texto[], char piglatin[]);
void escribirsalida(char piglatin[]);

main()
{
    char texto[80], piglatin[80];
    int palabras;

    printf("Bienvenido al generador de pig-latin\n\n");
    printf("Escribir \'FIN\' para terminar\n\n");

    do { /* procesar una nueva línea de texto */

        inicializar(texto, piglatin);
        leerentrada(texto);

        /* comprobación condición de finalización */
        if (toupper(texto[0]) == 'F' &&
            toupper(texto[1]) == 'I' &&
            toupper(texto[2]) == 'N') break;

        /* contar el número de palabras en la línea */
        palabras = contarpalabras(texto);

        /* convertir texto en "pig-latin" */
        convertir(palabras, texto, piglatin);
        escribirsalida(piglatin);
    }
```

```
   while (palabras >= 0);

   printf("\nueQa engata nua uenba iada (Que tenga un buen día)\n");
}

/* inicializar el array de caracteres con espacios en blanco */

void inicializar(char texto[], char piglatin[])
{
   int cont;

   for (cont = 0; cont < 80; ++cont)
       texto[cont] = piglatin[cont] = ' ';
   return;
}

/* leer una línea de texto */

void leerentrada(char texto[])

{
   int cont = 0;
   char c;

   while ((c = getchar()) != '\n') {
         texto[cont] = c;
         ++cont;
   }
   return;
}

/* examinar la línea de texto y contar el número de palabras */

int contarpalabras(char texto[])

{
   int cont, palabras = 1;

   for (cont = 0; cont < 79; ++cont)
       if (texto[cont] == ' ' && texto[cont + 1] != ' ')
          ++palabras;
   return(palabras);
}

/* convertir cada palabra en "pig-latin" */

void convertir(int palabras, char texto[], char piglatin[])
```

```
{
   int n, cont;
   int m1= 0;          /* indicador -> comienzo de la palabra */
   int m2;             /* indicador -> final de la palabra */

   /* convertir cada palabra */
   for (n = 1; n <= palabras; ++n) {

        /* localizar el final de la palabra actual */
        cont = m1;
        while (texto[cont] != ' ')
              m2 = cont++;

        /* transponer la primera letra y añadir una 'a' */
        for (cont = m1; cont < m2; ++cont)
            piglatin[cont + (n - 1)] = texto[cont + 1];
        piglatin[m2 + (n - 1)] = texto[m1];
        piglatin[m2 + n] = 'a';

        /* reinicializar el indicador inicial */
        m1 = m2 + 2;
   }
   return;
}

/* escribir la línea de texto en "pig-latin" */

void escribirsalida(char piglatin[])

{
   int cont = 0;

   for (cont = 0; cont < 80; ++cont)
        putchar(piglatin[cont]);
   printf("\n");
   return;
}
```

Observe que cada función requiere por lo menos un array como argumento. En `contarpalabras` y `escribirsalida` los argumentos de array simplemente proporcionan la entrada de la función. Sin embargo, en `convertir`, uno de los argumentos de array proporciona la entrada y el otro la salida a `main`. Y en `inicializar` y `leerentrada` los arrays representan información que es devuelta a `main`.

Las declaraciones de las funciones dentro de `main` están escritas como prototipos completos de funciones. Observe que cada argumento de array se identifica mediante un par de corchetes tras el nombre del array.

Ahora consideremos qué pasa cuando se ejecuta el programa. A continuación se muestra una sesión interactiva típica, donde las respuestas del usuario están subrayadas.

```
Bienvenido al generador de pig-latin

Escribir 'FIN' para terminar

C es un popular lenguaje de programacion estructurada
Ca sea nua opularpa enguajela eda rogramacionpa structuradaea

el beisbol es el gran pasatiempo americano,
lea eisbolba sea lea ranga asatiempopa mericano,aa

aunque hay muchos que prefieren el futbol
unqueaa ayha uchosma ueqa refierenpa lea utbolfa

por favor, no estornude en la sala de computadoras
ropa avor,fa ona stornudeea nea ala alasa eda omputadorasca

fin

ueQa engata nua uenba iada (Que tenga un buen día)
```

El programa no incluye ningún procesamiento especial de los signos de puntuación, las letras mayúsculas o los sonidos de doble letra (por ejemplo «qu» o «ll»). Estos refinamientos se dejan como ejercicio para el lector.

9.4. ARRAYS MULTIDIMENSIONALES

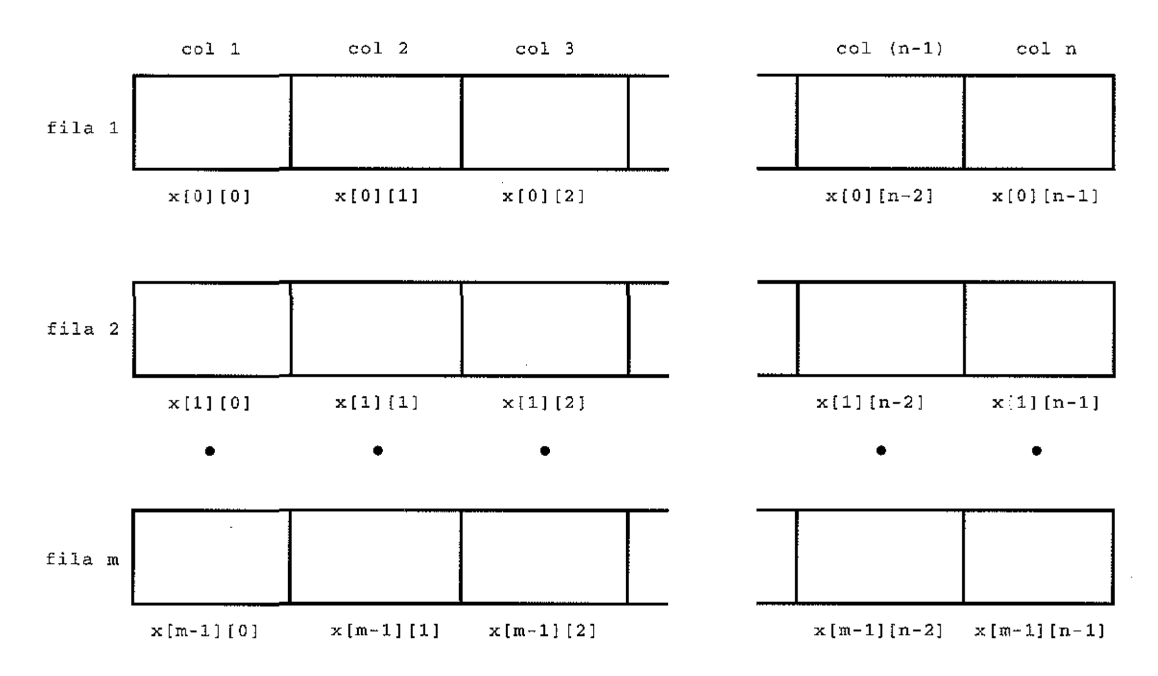
Los arrays multidimensionales se definen prácticamente de la misma manera que los arrays unidimensionales, excepto que se requiere un par separado de corchetes para cada índice. Así, un array bidimensional requerirá dos pares de corchetes, un array tridimensional requerirá tres pares de corchetes, y así sucesivamente.

En términos generales, la definición de un array multidimensional se puede escribir como

```
tipo-de-almacenamiento  tipo-dato  array[expresión 1]
                                   [expresión 2]...[expresión n];
```

donde *tipo-de-almacenamiento* se refiere al tipo de almacenamiento del array, *tipo-dato* es su tipo de datos, *array* el nombre del array, y *expresión 1*, *expresión 2*, ..., *expresión n* son expresiones enteras positivas que indican el número de elementos del array asociados con cada índice. Recordar que *tipo-de-almacenamiento* es opcional; los valores por omisión son `auto` para los arrays definidos dentro de una función y `extern` para los arrays definidos fuera de una función.

Hemos observado que un array unidimensional de `n` elementos puede ser visto como una *lista* de valores, como se ilustró en la Figura 9.1. Análogamente, un array bidimensional de `m` × `n` puede ser visto como una *tabla* de valores que tienen `m` filas y `n` columnas, como se ilustra en la Figura 9.2. Extendiendo esta idea, un array tridimensional puede verse como un *conjunto* de tablas (por ejemplo, un libro en el cual cada página es una tabla), y así sucesivamente.



x es un array bidimensional de m × n

Figura 9.2.

EJEMPLO 9.15. Varias definiciones de arrays multidimensionales se muestran a continuación.

```
float tabla[50][50];

char pagina[24][80];

static double registros[100][66][255];

static double registros[L][M][N];
```

La primera línea define `tabla` como un array de elementos en coma flotante con 50 filas y 50 columnas (por tanto 50 × 50 = 2500 elementos), y la segunda línea establece `pagina` como un array de caracteres con 24 filas y 80 columnas (24 × 80 = 1920 elementos). El tercer array puede ser visto como un conjunto de 100 tablas estáticas en doble precisión, cada una con 66 líneas y 255 columnas (por tanto 100 × 66 × 255 = 1 683 000 elementos).

La última definición es análoga a la definición precedente excepto que las constantes simbólicas `L`, `M` y `N` definen el tamaño del array. Así los valores asignados a dichas constantes simbólicas determinan el tamaño real del array.

Se debe tener cuidado en el orden en que los valores iniciales se asignan a los elementos del array multidimensional. (Recordar que *sólo pueden inicializarse los arrays estáticos y externos*.) La regla es que el último índice (extremo derecho) es el que se incrementa más rápidamente, y el primer índice (extremo izquierdo) es el que se incrementa más lentamente. Así los elementos de un array bidimensional deben ser asignados por filas, esto es, primero serán asignados los elementos de la primera fila, luego los elementos de la segunda, y así sucesivamente.

EJEMPLO 9.16. Considerar la siguiente definición de array bidimensional:

```
int valores[3][4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
```

Observe que `valores` puede ser visto como una tabla de tres filas y cuatro columnas (cuatro elementos por fila). Como los valores iniciales se asignan por filas (el último índice se incrementa más rápidamente), el resultado de esta asignación inicial será como sigue:

```
valores[0][0]=1   valores[0][1]=2    valores[0][2]=3    valores[0][3]=4
valores[1][0]=5   valores[1][1]=6    valores[1][2]=7    valores[1][3]=8
valores[2][0]=9   valores[2][1]=10   valores[2][2]=11   valores[2][3]=12
```

Recordar que el primer índice varía entre `0` y `2`, y el segundo entre `0` y `3`.

El orden natural en el que los valores iniciales son asignados se puede alterar formando grupos de valores iniciales encerrados entre llaves (`{...}`). Los valores dentro de cada par interno de llaves serán asignados a los elementos del array cuyo último índice varíe más rápidamente. Por ejemplo, en un array bidimensional, los valores almacenados dentro del par interno de llaves serán asignados a los elementos de una fila, ya que el segundo índice (columna) se incrementa más rápidamente. Si hay pocos elementos dentro de cada par de llaves, al resto de los elementos de cada fila se le asignarán ceros. Por otra parte, el número de valores dentro de cada par de llaves no puede exceder del tamaño de fila definido.

EJEMPLO 9.17. A continuación se muestra una variación de la definición de array bidimensional presentada en el último ejemplo.

```
int valores[3][4] = {
                        {1, 2, 3, 4},
                        {5, 6, 7, 8},
                        {9, 10, 11, 12}
                    };
```

Esta definición produce las mismas asignaciones iniciales que el último ejemplo. Así los cuatro valores del primer par de llaves internas son asignados a los elementos de la primera fila del array, los valores del segundo par de llaves son asignados a los elementos de la segunda fila del array, y así sucesivamente. Observe que el par externo de llaves se necesita para contener los pares internos.

Considerar ahora la definición del siguiente array bidimensional:

```
int valores[3][4] = {
                        {1, 2, 3},
                        {4, 5, 6},
                        {7, 8, 9}
                    };
```

Esta definición asigna valores solo a los tres primeros elementos en cada fila. Por tanto, los elementos del array tendrán los siguientes valores iniciales:

```
valores[0][0]=1    valores[0][1]=2    valores[0][2]=3    valores[0][3]=0
valores[1][0]=4    valores[1][1]=5    valores[1][2]=6    valores[1][3]=0
valores[2][0]=7    valores[2][1]=8    valores[2][2]=9    valores[2][3]=0
```

Observe que el último elemento en cada fila tiene asignado valor cero.

Si la anterior definición de array se escribe como

```
int valores[3][4] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
```

entonces tres elementos del array tendrán asignado el valor cero, pero el orden de asignación será diferente. En particular, los elementos del array tendrán los siguientes valores iniciales:

```
valores[0][0]=1    valores[0][1]=2    valores[0][2]=3    valores[0][3]=4
valores[1][0]=5    valores[1][1]=6    valores[1][2]=7    valores[1][3]=8
valores[2][0]=9    valores[2][1]=0    valores[2][2]=0    valores[2][3]=0
```

Ahora los valores iniciales se asignan fila a fila, con el último índice que se incrementa más rápidamente, hasta que todos los valores iniciales hayan sido especificados. Sin embargo, sin los pares internos de llaves no se pueden agrupar los valores iniciales para asignarlos a una fila específica.

Finalmente, considerar la definición del array

```
int valores[3][4] = {
                        {1, 2, 3, 4, 5},
                        {6, 7, 8, 9, 10},
                        {11, 12, 13, 14, 15}
                    };
```

Esto producirá un error de compilación, ya que el número de valores dentro de cada par interno de llaves (cinco valores en cada par) excede el tamaño definido del array (cuatro elementos en cada fila).

El uso de grupos internos de valores iniciales se puede generalizar para arrays de mayor dimensión.

EJEMPLO 9.18. Considerar la siguiente definición de array tridimensional:

```
int t[10][20][30] = {
                        {                       /* tabla 1 */
                            {1, 2, 3, 4},       /* fila 1 */
                            {5, 6, 7, 8},       /* fila 2 */
                            {9, 10, 11, 12}     /* fila 3 */
                        },
                        {                       /* tabla 2 */
                            {21, 22, 23, 24},   /* fila 1 */
                            {25, 26, 27, 28},   /* fila 2 */
                            {29, 30, 31, 32}    /* fila 3 */
                        }
                    };
```

Ver este array como una colección de 10 tablas, con 20 filas y 30 columnas. El grupo de valores iniciales producirá la asignación de los siguientes valores no nulos en las primeras dos tablas.

```
t[0][0][0]=1      t[0][0][1]=2      t[0][0][2]=3      t[0][0][3]=4
t[0][1][0]=5      t[0][1][1]=6      t[0][1][2]=7      t[0][1][3]=8
t[0][2][0]=9      t[0][2][1]=10     t[0][2][2]=11     t[0][2][3]=12

t[1][0][0]=21     t[1][0][1]=22     t[1][0][2]=23     t[1][0][3]=24
t[1][1][0]=25     t[1][1][1]=26     t[1][1][2]=27     t[1][1][3]=28
t[1][2][0]=29     t[1][2][1]=30     t[1][2][2]=31     t[1][2][3]=32
```

El resto de los elementos del array tendrán asignados ceros.

Los arrays multidimensionales se procesan de la misma manera que los arrays unidimensionales, sobre la base de elemento a elemento. Sin embargo, se requiere algún cuidado cuando se pasan arrays multidimensionales a una función. En particular, las declaraciones de argumentos formales dentro de la definición de función *deben* incluir especificaciones explícitas de tamaño en todos los índices *excepto en el primero*. Estas especificaciones deben ser consistentes con las correspondientes especificaciones de tamaño en el programa que hace la llamada. El primer índice puede ser escrito como un par de corchetes vacíos, como en un array unidimensional. Los prototipos correspondientes de función deben escribirse de la misma manera.

EJEMPLO 9.19. Suma de dos tablas de números. Supongamos que queremos leer dos tablas de enteros en la computadora, calcular la suma de los elementos correspondientes, esto es,

```
c[i][j] = a[i][j] + b[i][j]
```

y a continuación escribir la nueva tabla que contiene estas sumas. Supondremos que todas las tablas contendrán el mismo número de filas y de columnas, no excediendo de 20 filas y 30 columnas.

Haremos uso de las siguientes definiciones de arrays y variables.

- `a, b, c` = arrays bidimensionales con el mismo número de filas y el mismo número de columnas, no excediendo de 20 filas y 30 columnas.
- `nfilas` = variable entera que indica el número real de filas en cada tabla.
- `ncols` = variable entera que indica el número real de columnas en cada tabla.
- `fila` = contador entero que indica el número de fila.
- `col` = contador entero que indica el número de columna.

El programa se modularizará escribiendo funciones individuales para leer el array, calcular la suma de los elementos del array y escribir el array. Llamaremos a estas funciones `leerentrada`, `calcularsuma` y `escribirsalida`, respectivamente.

La lógica dentro de cada función es bastante directa. A continuación se muestra el programa completo en C para realizar el cálculo.

```
/* calcular la suma de los elementos en dos tablas de enteros */

#include <stdio.h>

#define MAXFIL 20
#define MAXCOL 30
```

```
void leerentrada(int a[][MAXCOL], int nfilas, int ncols);
void calcularsuma(int a[][MAXCOL], int b[][MAXCOL],
                  int c[][MAXCOL], int nfilas, int ncols);
void escribirsalida(int c[][MAXCOL], int nfilas, int ncols);

main()

{
  int nfilas, ncols;

  /* definiciones de arrays */
  int a[MAXFIL][MAXCOL], b[MAXFIL][MAXCOL], c[MAXFIL][MAXCOL];

  printf("¿Cuantas filas? ");
  scanf("%d", &nfilas);
  printf("¿Cuantas columnas? ");
  scanf("%d", &ncols);

  printf("\n\nPrimera tabla:\n");
  leerentrada(a, nfilas, ncols);

  printf("\n\nSegunda tabla:\n");
  leerentrada(b, nfilas, ncols);

  calcularsuma(a, b, c, nfilas, ncols);

  printf("\n\nSumas de los elementos:\n\n");
  escribirsalida(c, nfilas, ncols);
}

/* leer una tabla de enteros */

void leerentrada(int a[][MAXCOL], int m, int n)

{
   int fila, col;

   for (fila = 0; fila < m; ++fila) {
       printf("\nIntroducir datos para la fila nº %2d\n", fila + 1);
       for (col = 0; col < n; ++col)
           scanf("%d", &a[fila][col]);
   }
   return;
}

/* sumar los elementos de dos tablas de enteros */

void calcularsuma(int a[][MAXCOL], int b[][MAXCOL],
                  int c[][MAXCOL], int m, int n)
{
  int fila, col;
```

```
    for (fila = 0; fila < m; ++fila)
        for (col = 0; col < n; ++col)
            c[fila][col] = a[fila][col] + b[fila][col];
    return;
}

/* escribir una tabla de enteros */

void escribirsalida(int a[][MAXCOL], int m, int n)

{
    int fila, col;

    for (fila = 0; fila < m; ++fila) {
        for (col = 0; col < n; ++col)
            printf("%4d", a[fila][col]);
        printf("\n");
    }
    return;
}
```

Las definiciones de arrays se expresan en términos de las constantes simbólicas `MAXFIL` y `MAXCOL`, cuyos valores se especifican como `20` y `30`, respectivamente, al principio del programa.

Observe la manera de escribir las declaraciones de los argumentos formales dentro de cada definición de función. Por ejemplo, la primera línea de la función `leerentrada` se escribe como

```
void leerentrada(int a[][MAXCOL], int m, int n)
```

El nombre del array, `a`, está seguido por dos pares de corchetes. El primer par está vacío porque el número de filas no necesita ser especificado explícitamente. Sin embargo, el segundo par contiene la constante simbólica `MAXCOL`, que provee una especificación de tamaño explícita para el número de columnas. Los nombres de arrays que aparecen en otras definiciones de función (por ejemplo, en las funciones `calcularsuma` y `escribirsalida`) están escritos de la misma manera.

Observe también los prototipos de funciones al comienzo del programa. Cada prototipo es análogo a la primera línea de la definición de función correspondiente. En particular, cada nombre de array está seguido por dos pares de corchetes, el primero de los cuales está vacío. El segundo par de corchetes contiene la especificación de tamaño requerida para el número de columnas.

Supongamos que el programa se utiliza para sumar las dos tablas de números siguientes:

Primera tabla				*Segunda tabla*			
1	2	3	4	10	11	12	13
5	6	7	8	14	15	16	17
9	10	11	12	18	19	20	21

La ejecución del programa generará el siguiente diálogo. (Como siempre, las respuestas del usuario están subrayadas.)

```
¿Cuantas filas? 3
¿Cuantas columnas? 4

Primera tabla:

Introducir datos para la fila nº  1
1 2 3 4

Introducir datos para la fila nº  2
5 6 7 8

Introducir datos para la fila nº  3
9 10 11 12

Segunda tabla:

Introducir datos para la fila nº  1
10 11 12 13

Introducir datos para la fila nº  2
14 15 16 17

Introducir datos para la fila nº  3
18 19 20 21

Sumas de los elementos:

11  13  15  17
19  21  23  25
27  29  31  33
```

En algunos compiladores de C no es posible pasar arrays multidimensionales de distinto tamaño a funciones. En tales situaciones es posible rediseñar el programa de modo que los arrays multidimensionales se definan externamente (globales). Por tanto, los arrays no necesitan ser pasados a las funciones como argumentos. Sin embargo, esta estrategia no funcionará siempre, ya que algunos programas (tales como el programa mostrado en el último ejemplo) utilizan la misma función para procesar diferentes arrays. Los problemas de este tipo pueden ser evitados mediante el uso de punteros, como se discutirá en el próximo capítulo.

9.5. ARRAYS Y CADENAS DE CARACTERES

Ya hemos visto que una cadena de caracteres puede ser representada por un array unidimensional de caracteres. Cada carácter de la cadena será almacenado en un elemento del array. Algunos problemas requieren que los caracteres de la cadena sean procesados individualmente (por ejemplo, el generador de «pig latin» mostrado en el Ejemplo 9.14). No obstante, hay muchos otros problemas en los que se requiere que las cadenas de caracteres se procesen como entidades completas. Tales problemas pueden simplificarse considerablemente utilizando funciones especiales de biblioteca orientadas a cadenas de caracteres.

Por ejemplo, la mayoría de los compiladores de C incluyen funciones de biblioteca que permiten comparar cadenas de caracteres, copiarlas o *concatenarlas* (es decir, combinarlas una detrás de otra). Otras funciones de biblioteca permiten operaciones sobre caracteres individuales dentro de las cadenas; por ejemplo, permiten encontrar caracteres individuales dentro de una cadena, y así sucesivamente. El siguiente ejemplo ilustra el uso de algunas de estas funciones de biblioteca.

EJEMPLO 9.20. Reordenación de una lista de cadenas de caracteres. Supongamos que queremos leer una lista de cadenas de caracteres, reordenarlas alfabéticamente y escribir la lista reordenada. La estrategia para hacer esto es muy parecida a la mostrada en el Ejemplo 9.13, donde se reordenó una lista de números en orden ascendente. Sin embargo, ahora existe la complicación adicional de comparar cadenas de caracteres completas, en lugar de valores numéricos simples. Por tanto, almacenaremos las cadenas de caracteres en un array bidimensional de caracteres. Cada cadena se almacenará en una fila distinta del array.

Para simplificar el proceso, hacemos uso de las funciones de biblioteca `strcmp` y `strcpy`. Estas funciones se utilizan para comparar dos cadenas de caracteres y para copiar una cadena en otra, respectivamente. (Algunos compiladores también incluyen la función `strcmpi`, que es una variación de la más común `strcmp`. El uso de `strcmpi` es algunas veces más conveniente, ya que no distingue entre mayúsculas y minúsculas. Sin embargo, no está soportada por el estándar ANSI.)

La función `strcmp` acepta dos cadenas como argumentos y devuelve un valor entero, dependiendo del orden relativo de las dos cadenas, como sigue:

1. Se devuelve un valor negativo si la primera cadena precede alfabéticamente a la segunda.
2. Se devuelve cero si las dos cadenas son idénticas (caso no considerado).
3. Se devuelve un valor positivo si la segunda cadena precede alfabéticamente a la primera.

Por tanto, si `strcmp(cadena1, cadena2)` devuelve un valor positivo, indicará que la `cadena2` debe ir antes que la `cadena1` para colocar las cadenas en orden alfabético.

La función `strcpy` acepta también dos cadenas como argumentos. Generalmente su primer argumento es un identificador que representa una cadena. El segundo argumento puede ser una cadena constante o un identificador que represente una cadena. La función copia el valor de `cadena2` en `cadena1`. Por tanto, esto hace que una cadena sea asignada a otra.

El programa completo es muy similar al programa de reordenación de números presentado en el Ejemplo 9.13. Sin embargo, ahora se permite que el programa acepte un número no especificado de cadenas, hasta que se introduzca una cadena cuyos tres primeros caracteres sean `FIN` (tanto en minúsculas como en mayúsculas). El programa contará las cadenas según sean introducidas, ignorando la última cadena que contiene `FIN`.

A continuación se muestra el programa completo.

```
/* ordenar alfabéticamente una lista de cadenas de caracteres,
   utilizando un array bidimensional */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void reordenar(int n, char x[][12]);   /* prototipo de función */

main()
{
   int i, n = 0;
   char x[10][12];
```

```
    printf("Introducir debajo cada cadena en una línea\n\n");
    printf("Escribir \'FIN\' para terminar\n\n");

    /* leer la lista de cadenas de caracteres */
    do {
          printf("cadena %d: ", n + 1);
          scanf("%s", x[n]);
       } while (strcmp(x[n++], "FIN"));

    /* ajustar el valor de n */

    n--;
    /* reordenar la lista de cadenas de caracteres */
    reordenar(n, x);

    /* escribir la lista reordenada de cadenas de caracteres */
    printf("\n\nLista reordenada de cadenas:\n");
    for (i = 0; i < n; ++i)
        printf("\ncadena %d: %s", i + 1, x[i]);
}

void reordenar(int n, char x[][12])  /* reordenar la lista de ca-
                                        denas de caracteres */

{
    char temp[12];
    int i, elem;

    for (elem = 0; elem < n - 1; ++elem)

        /* encontrar la menor de las cadenas restantes */
        for (i = elem + 1; i < n; ++i)

            if (strcmp(x[elem], x[i]) > 0) {
                /* intercambiar las dos cadenas */
                strcpy(temp, x[elem]);
                strcpy(x[elem], x[i]);
                strcpy(x[i], temp);
            }
    return;
}
```

La función `strcmp` aparece en dos sitios distintos del programa: en `main`, cuando se comprueba la condición de finalización, y en `reordenar`, cuando se comprueba la necesidad de intercambiar dos cadenas. El intercambio real se efectúa usando `strcpy`.

A continuación se muestra un diálogo típico de la ejecución de este programa. Como siempre, las respuestas del usuario están subrayadas.

```
Introducir debajo cada cadena en una línea

Escribir 'FIN' para terminar
```

```
cadena 1: PACIFICO
cadena 2: ATLANTICO
cadena 3: INDICO
cadena 4: CARIBE
cadena 5: BERING
cadena 6: NEGRO
cadena 7: ROJO
cadena 8: NORTE
cadena 9: BALTICO
cadena 10: CASPIO
cadena 11: FIN

Lista reordenada de cadenas

cadena 1: ATLANTICO
cadena 2: BALTICO
cadena 3: BERING
cadena 4: CARIBE
cadena 5: CASPIO
cadena 6: INDICO
cadena 7: NEGRO
cadena 8: NORTE
cadena 9: PACIFICO
cadena 10: ROJO
```

En el próximo capítulo veremos diferentes maneras de representar listas de cadenas de modo más eficiente en términos de requerimientos de memoria.

CUESTIONES DE REPASO

9.1. ¿En qué se diferencia un array de una variable ordinaria?
9.2. ¿Qué condiciones deben cumplir todos los elementos de cualquier array?
9.3. ¿Cómo se identifican los elementos individuales de un array?
9.4. ¿Qué son los índices? ¿Cómo se escriben? ¿Qué restricciones se aplican a los valores de los índices?
9.5. Sugerir una forma práctica de visualizar arrays de una y de dos dimensiones.
9.6. ¿En qué se diferencia la definición de un array de la de una variable ordinaria?
9.7. Mencionar las reglas para escribir definiciones de arrays unidimensionales.
9.8. ¿Qué ventaja tiene definir el tamaño de un array en términos de una constante simbólica en vez de usar una cantidad entera fija?
9.9. ¿Pueden especificarse valores iniciales en una definición de array externo? ¿Pueden especificarse en una definición de array automático?
9.10. ¿Cómo se escriben los valores iniciales en una definición de un array unidimensional? ¿Debe inicializarse todo el array?
9.11. ¿Qué valor se les asigna automáticamente a los elementos del array que no están explícitamente inicializados?

9.12. Describir la forma habitual de asignar inicialmente una constante de cadena de caracteres a un array unidimensional. ¿Puede usarse un procedimiento similar para asignar valores iniciales a un array numérico?

9.13. Cuando a un array de caracteres unidimensional de tamaño no especificado se le asigna un valor inicial, ¿qué carácter extra se añade al final de la cadena?

9.14. ¿Cuándo se requieren declaraciones de arrays (en contraste con definiciones de arrays) en un programa en C? ¿En qué se diferencian estas declaraciones de las definiciones de arrays?

9.15. ¿Cómo se procesan generalmente los arrays en C? ¿Pueden procesarse los arrays con instrucciones simples, sin repetición?

9.16. Cuando se pasa un array a una función, ¿cómo se debe escribir el argumento de array? ¿Cómo se escribe el argumento formal correspondiente?

9.17. ¿Cómo se interpreta el nombre del array cuando se pasa a una función?

9.18. Supongamos que una declaración de función incluye las especificaciones de tipo, y uno de los argumentos es un array. ¿Cómo debe escribirse la especificación del tipo del array?

9.19. Cuando se pasa un argumento a una función, ¿cuál es la diferencia entre el paso por valor y el paso por referencia? ¿A qué tipo de argumentos se le debe aplicar cada uno?

9.20. Si se pasa un array a una función y dentro de ella se modifican varios de sus elementos, ¿se reconocen estos cambios en la parte del programa que hizo la llamada? Explicarlo.

9.21. ¿Se puede pasar un array desde una función hasta la parte del programa que hizo la llamada, mediante una instrucción `return`?

9.22. ¿Cómo se definen los arrays multidimensionales? Comparar con la forma de definir los arrays unidimensionales.

9.23. Enunciar la regla que determina el orden de asignación de valores iniciales a los elementos de un array multidimensional?

9.24. Cuando se asignan valores iniciales a los elementos de un array multidimensional, ¿qué ventajas existen al formar grupos de valores iniciales, donde cada grupo se encierra entre su propio conjunto de llaves?

9.25. Cuando se pasa un array multidimensional a una función, ¿cómo se escribe la declaración formal del argumento? Comparar con los arrays unidimensionales.

9.26. ¿Cómo puede almacenarse una lista de cadenas de caracteres en un array bidimensional? ¿Cómo pueden procesarse las cadenas individuales? ¿Qué funciones de biblioteca se tienen para simplificar el procesado de cadenas?

PROBLEMAS

9.27. Describir el array definido en cada una de las siguientes instrucciones:

a) `char nombre[30];`

b) `float c[6];`

c)
```
#define N 50
. . . . .
int a[N];
```

d) `int parametros[5][5];`

e)
```
#define A 66
#define B 132
. . . . .
char memo[A][B];
```

f)
```
double cuentas[50][20][80];
```

9.28. Describir el array definido en cada una de las siguientes instrucciones. Indicar qué valores son asignados a los elementos individuales del array.

a)
```
float c[8] = {2., 5., 3., -4., 12., 12., 0., 8.};
```

b)
```
float c[8] = {2., 5., 3., -4.};
```

c)
```
int z[12] = {0, 0, 8, 0, 0, 6};
```

d)
```
char indicador[9] = {'V', 'E', 'R', 'D', 'A', 'D', 'E', 'R', 'O',};
```

e)
```
char indicador[10] = {'V', 'E', 'R', 'D', 'A', 'D', 'E', 'R', 'O',}
```

f)
```
char indicador[] = "VERDADERO";
```

g)
```
char indicador[] = "FALSO";
```

h)
```
int p[2][4] = {1, 3, 5, 7};
```

i)
```
int p[2][4] = {1, 1, 3, 3, 5, 5, 7, 7};
```

j)
```
int p[2][4] = {
                {1, 3, 5, 7},
                {2, 4, 6, 8}
              };
```

k)
```
int p[2][4] = {
                {1, 3},
                {5, 7}
              };
```

l)
```
int c[2][3][4] = {
                   {
                     {1, 2, 3},
                     {4, 5},
                     {6, 7, 8, 9}
                   },
                   {
                     {10, 11},
                     {},
                     {12, 13, 14}
                   }
                 };
```

m)
```
char colores[3][6] = {
                       {'R', 'O', 'J', 'O'},
                       {'V', 'E', 'R', 'D', 'E'},
                       {'A', 'Z', 'U', 'L'}
                     };
```

9.29. Escribir una definición apropiada de array para cada uno de los siguientes problemas.

a) Definir un array unidimensional de 12 elementos enteros llamado `c`. Asignar los valores 1, 4, 7, 10, ..., 34 a los elementos del array.

b) Definir un array unidimensional de caracteres llamado `punto`. Asignar la cadena `"NORTE"` a los elementos del array. Terminar la cadena con el carácter nulo.

c) Definir un array unidimensional de cuatro caracteres llamado `letras`. Asignar los caracteres '`N`', '`S`', '`E`' y '`O`' a los caracteres del array.

d) Definir un array unidimensional de seis elementos en coma flotante llamado `constantes`. Asignar los siguientes valores a los elementos del array:

0.005 –0.032 1e – 6 0.167 –0.3e8 0.015

e) Definir un array bidimensional de enteros, de 3 × 4, llamado `n`. Asignar los siguientes valores iniciales a los elementos del array:

10	12	14	16
20	22	24	26
30	32	34	36

f) Definir un array bidimensional de enteros, de 3 × 4, llamado `n`. Asignar los siguientes valores iniciales a los elementos del array:

10	12	14	0
0	20	22	0
0	30	32	0

g) Definir un array bidimensional de enteros, de 3 × 4, llamado `n`. Asignar los siguientes valores iniciales a los elementos del array:

10	12	14	16
20	22	0	0
0	0	0	0

9.30. Para cada una de las siguientes situaciones, escribir las definiciones y declaraciones necesarias para transferir las variables y los arrays indicados desde `main` hasta la función llamada `muestra` (ver Ejemplos 9.10 y 9.11). En cada caso, asignar el valor devuelto por la función a la variable en coma flotante `x`.

a) Transferir las variables en coma flotante `a` y `b` y el array unidimensional de 20 elementos enteros `jstar`.

b) Transferir la variable entera `n`, la variable carácter `c` y el array unidimensional de 50 elementos en doble precisión `valores`.

c) Transferir el array bidimensional de caracteres de 12 × 80, llamado `texto`.

d) Transferir el array unidimensional de 40 caracteres `mensaje` y el array bidimensional de 50 × 100 elementos en coma flotante `cuentas`.

9.31. Describir la salida producida por los siguientes programas:

a)
```
#include <stdio.h>

main()
{
```

```
    int a, b = 0;
    static int c[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};

    for (a = 0; a < 10; ++a)
        if ((c[a] % 2) == 0)  b+= c[a];
    printf("%d", b);
}
```

b)
```
#include <stdio.h>

main()
{
    int a, b = 0;
    static int c[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};

    for (a = 0; a < 10; ++a)
        if ((a % 2) == 0)  b+= c[a];
    printf("%d", b);
}
```

c)
```
#include <stdio.h>

main()
{
    int a, b = 0;
    int c[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};

    for (a = 0; a < 10; ++a)
        b+= c[a];
    printf("%d", b);
}
```

d)
```
#include <stdio.h>

int c[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};

main()
{
    int a, b = 0;

    for (a = 0; a < 10; ++a)
        if ((c[a] % 2) == 1)  b+= c[a];
    printf("%d", b);
}
```

e)
```
#include <stdio.h>

#define FILAS 3
#define COLUMNAS 4

int z[FILAS][COLUMNAS] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
```

```
main()
{
     int a, b, c = 999;

     for (a = 0; a < FILAS; ++a)
          for (b = 0; b < COLUMNAS; ++b)
               if (z[a][b] < c)  c = z[a][b];
     printf("%d",  c);
   }
```

f)

```
#include <stdio.h>

#define FILAS 3
#define COLUMNAS 4

int z[FILAS][COLUMNAS] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};

main()
{
     int a, b, c;

     for (a = 0; a < FILAS; ++a) {
          c = 999;
          for (b = 0; b < COLUMNAS; ++b)
               if (z[a][b] < c)  c = z[a][b];
          printf("%d ",  c);
     }
}
```

g)

```
#include <stdio.h>

#define FILAS 3
#define COLUMNAS 4

void sub1(int z[][COLUMNAS]);

main()
{
     static int z[FILAS][COLUMNAS] = {1, 2, 3, 4, 5, 6, 7, 8, 9,
                                                  10, 11, 12};

     sub1(z);
}

void sub1(int x[][4])
{
   int a, b, c;

   for (b = 0; b < COLUMNAS; ++b) {
        c = 0;
        for (a = 0; a < FILAS; ++a)
             if (x[a][b] > c)   c = x[a][b];
        printf("%d  ", c);
   }
   return;
}
```

h)
```
#include <stdio.h>

#define FILAS 3
#define COLUMNAS 4

void sub1(int z[][COLUMNAS]);

main()
{
     int a, b;
     static int z[FILAS][COLUMNAS] = {1, 2, 3, 4, 5, 6, 7, 8, 9,
                                       10, 11, 12};

     sub1(z);

     for (a = 0; a < FILAS; ++a) {
          for (b = 0; b < COLUMNAS; ++b)
               printf("%d  ", z[a][b]);
          printf("\n");
     }
}

void sub1(int x[][COLUMNAS])
{
     int a, b;

     for (a = 0; a < FILAS; ++a)
          for (b = 0; b < COLUMNAS; ++b)
               if ((x[a][b] % 2) == 1) x[a][b]--;
     return;
}
```

i)
```
#include <stdio.h>

main()
{
     int a;
     static char c[] = "Programar en C puede ser muy divertido!";

     for (a = 0; c[a] != '\0'; ++a)
          if ((a % 2) == 0)
               printf("%c%c", c[a], c[a]);
}
```

PROBLEMAS DE PROGRAMACIÓN

9.32. Modificar el programa dado en el Ejemplo 9.8 (desviaciones respecto de la media) para incluir dos funciones adicionales. La primera función lee los números para calcular la media y a la vez realiza la suma. La segunda función debe calcular las desviaciones respecto a la media. El resto de las

acciones (leer el valor de `n`, calcular el valor de la media, escribir la media calculada y las desviaciones respecto de ella) deben realizarse en la parte `main` del programa.

9.33. Modificar el programa dado en el Ejemplo 9.9 (desviaciones respecto a la media, revisión) incluyendo dos funciones adicionales. Calcular y escribir la media en la primera función. Calcular y escribir las desviaciones respecto de la media en la segunda función.

9.34. Modificar el programa dado en el Ejemplo 9.13 (reordenación de una lista de números) de modo que los números se reordenen en secuencia de valores *decrecientes* (del mayor al menor). Comprobar el programa con los datos dados en el Ejemplo 9.13.

9.35. Modificar el programa dado en el Ejemplo 9.13 (reordenación de una lista de números) de modo que se puedan realizar cualquiera de las siguientes reordenaciones:

a) de menor a mayor, en valor absoluto
b) de menor a mayor, algebraicamente (con signo)
c) de mayor a menor, en valor absoluto
d) de mayor a menor, algebraicamente (con signo)

Incluir un menú que permita al usuario seleccionar qué reordenación será usada cada vez que se ejecute el programa. Comprobar el programa usando los diez valores siguientes:

4.7	–8.0
–2.3	11.4
12.9	5.1
8.8	–0.2
6.0	–14.7

9.36. Modificar el generador de «pig latin» dado en el Ejemplo 9.14 de modo que permita las marcas de puntuación, letras mayúsculas y sonidos de letras dobles.

9.37. Modificar el programa dado en el Ejemplo 9.19 (suma de dos tablas de números) de modo que calcule las diferencias en vez de las sumas de los elementos correspondientes en las dos tablas de enteros. Comprobar el programa usando los datos dados en el Ejemplo 9.19.

9.38. Modificar el programa dado en el Ejemplo 9.19 (suma de dos tablas de números) de modo que se utilice un array tridimensional en vez de tres arrays bidimensionales. El primer índice referirá a una de las tres tablas. El segundo referirá el número de fila y el tercero el número de columna.

9.39. Escribir un programa en C que lea una línea de texto, la almacene en un array y la escriba al revés. La longitud de la línea no será especificada (terminará al pulsar la tecla `Intro`), pero se supone que no excederá de 80 caracteres.
Comprobar el programa con cualquier línea de texto de su elección. Comparar con el programa dado en el Ejemplo 7.15, que hace uso de la recursividad en vez de utilizar un array. ¿Qué enfoque es mejor y por qué?

9.40. Escribir un programa interactivo en C para procesar las notas de un grupo de estudiantes de un curso de programación en C. Empezar especificando el número de notas de examen para cada estudiante (suponer que este valor es el mismo para todos los estudiantes de la clase). Después introdu-

cir el nombre de cada estudiante y las notas de los exámenes. Calcular una nota media para cada estudiante y una media general de la clase (una media de las medias de los estudiantes). Escribir la media de la clase, seguida por el nombre, las notas individuales de los exámenes y la media para cada estudiante.

Almacenar los nombres de los estudiantes en un array bidimensional de caracteres y las notas en un array bidimensional en coma flotante. Hacer el programa lo más general posible. Rotular claramente la salida.

Comprobar el programa utilizando el siguiente conjunto de notas de exámenes de estudiantes.

Nombre			*Puntuaciones*			
Adrián	45	80	80	95	55	75
Antonio	60	50	70	75	55	80
Carmen	40	30	10	45	60	55
Félix	0	5	5	0	10	5
Guillermo	90	85	100	95	90	90
José	95	90	80	95	85	80
Lidia	35	50	55	65	45	70
Maruja	75	60	75	60	70	80
Pedro	85	75	60	85	90	100
Raúl	50	60	50	35	65	70
Rosa	70	60	75	70	55	75
Sonia	10	25	35	20	30	10
Víctor	25	40	65	75	85	95
Zoraida	65	80	70	100	60	95

Comparar con el programa escrito para el Problema 6.69(*k*).

9.41. Modificar el programa escrito para el problema anterior de manera que permita pesos desiguales en las notas de los exámenes individuales. En particular, suponer que cada uno de los primeros cuatro exámenes contribuye con el 15 por 100 de la nota final, y cada uno de los dos últimos con el 20 por 100 [ver Problema 6.69(l)].

9.42. Extender el programa escrito para el problema anterior de modo que se determine la desviación de la media de cada estudiante respecto de la media general. Escribir la media de la clase, seguida por cada nombre de estudiante, las notas de los exámenes individuales, la nota final y la desviación respecto de la media general. Asegurarse de que la salida se organiza de forma lógica y se rotula claramente.

9.43. Escribir un programa en C que genere una tabla de valores para la ecuación

$$y = 2e^{-0.1t} \text{ seno } 0.5t$$

donde t varía entre 0 y 60. Permitir que el valor del incremento t sea introducido como un parámetro de entrada.

9.44. Escribir un programa completo en C que genere una tabla de factores de interés compuesto, *F/P*, donde

$$F/P = (i + i/100)^n$$

En esta fórmula *F* representa el valor futuro de una suma dada de dinero, *P* su valor actual, *i* la tasa de interés anual, expresada como un porcentaje, y *n* el número de años.

Cada fila en la tabla corresponde a un valor diferente de *n*, con *n* en el rango de 1 a 30 (por tanto, 30 filas). Cada columna representa una tasa de interés diferente. Incluir las siguientes tasas de interés: 4, 4.5, 5, 5.5, 6, 6.5, 7, 7.5, 8, 8.5, 9, 9.5, 10, 11, 12 y 15 por ciento (por tanto, un total de 16 columnas). Asegurarse de rotular las filas y las columnas de forma apropiada.

9.45. Considerar las siguientes monedas extranjeras y sus equivalencias en dólares USA:

Libra británica:	0.65 libras por dólar USA
Dólar canadiense:	1.4 dólares por dólar USA
Florín holandés:	1.7 florines por dólar USA
Franco francés:	5.3 francos por dólar USA
Marco alemán:	1.5 marcos por dólar USA
Lira italiana:	1570 liras por dólar USA
Yen japonés:	98 yenes por dólar USA
Peso mejicano:	3.4 pesos por dólar USA
Franco suizo:	1.3 francos por dólar USA

Escribir un programa interactivo, guiado por menús, que acepte dos monedas extranjeras y devuelva el valor de la segunda moneda por cada unidad de la primera moneda. (Por ejemplo, si las dos monedas son el yen japonés y el peso mejicano, el programa devolverá el número de pesos mejicanos equivalentes a un yen japonés.) Utilizar los datos dados anteriores para realizar las conversiones. Diseñar el programa de modo que se ejecute repetidamente, hasta que se seleccione la condición de salida del menú.

9.46. Considerar la siguiente lista de países y sus capitales:

Canadá	Ottawa
Inglaterra	Londres
Francia	París
Alemania	Bonn
India	Nueva Delhi
Israel	Jerusalén
Italia	Roma
Japón	Tokio
México	Ciudad de México
República Popular China	Pekín
Rusia	Moscú
Estados Unidos	Washington

Escribir un programa interactivo en C que acepte el nombre de un país como entrada y escriba su correspondiente capital y viceversa. Diseñar el programa de modo que se ejecute repetidamente, hasta que se introduzca la palabra `Fin`.

9.47. Escribir un programa completo en C para cada uno de los problemas presentados a continuación. Incluir el tipo de arrays más apropiado para cada problema. Asegurarse de modularizar cada programa, rotular claramente la salida y hacer uso de tipos de datos normales y de estructuras eficientes de control.

a) Suponer que tenemos una tabla de enteros A, com m filas y n columnas, y una lista de enteros, X, con n elementos. Se desea generar una nueva lista de enteros, Y, que se forma realizando las siguientes operaciones:

```
y[1] = A[1][1] * X[1] + A[1][2] * X[2] + . . . + A[1][n] * X[n]
y[2] = A[2][1] * X[1] + A[2][2] * X[2] + . . . + A[2][n] * X[n]
. . . . .
y[m] = A[m][1] * X[1] + A[m][2] * X[2] + . . . + A[m][n] * X[n]
```

Escribir los datos de entrada (los valores de los elementos A y X), seguidos por los valores de los elementos de Y.

Usar el programa para procesar los siguientes datos:

$$A=\begin{bmatrix}1&2&3&4&5&6&7&8\\2&3&4&5&6&7&8&9\\3&4&5&6&7&8&9&10\\4&5&6&7&8&9&10&11\\5&6&7&8&9&10&11&12\\6&7&8&9&10&11&12&13\end{bmatrix}\qquad X=\begin{bmatrix}1\\-8\\3\\-6\\5\\-4\\7\\-2\end{bmatrix}$$

b) Suponer que A es una tabla de números en coma flotante, que tiene k filas y m columnas, y B es una tabla de números en coma flotante de m filas y n columnas. Se desea generar una nueva tabla C, donde cada elemento se determina mediante

```
C[i][j] = A[i][1] * B[1][j] + A[i][2] * B[2][j] + . . . +
                    A[i][m] * B[m][j]
```

donde i=1,2,...,k y j=1,2,...,n. (Esta operación es la *multiplicación de matrices*.)

Utilizar el programa para procesar los siguientes datos:

$$A=\begin{bmatrix}2&-1/3&0&2/3&4\\1/2&3/2&4&-2&1\\0&3&-9/7&6/7&4/3\end{bmatrix}\qquad B=\begin{bmatrix}6/5&0&-2&1/3\\5&7/2&3/4&-3/2\\0&-1&1&0\\9/2&3/7&-3&3\\4&-1/2&0&3/4\end{bmatrix}$$

Escribir los elementos de A, B y C. Asegurarse de que todo esté bien rotulado.

c) Leer los primeros m elementos de un array unidimensional en coma flotante. Calcular la suma de estos elementos, la media, las desviaciones, la desviación estándar, el máximo y el mínimo algebraico.

La *media* se define como

$$x_{media} = (x_1 + x_2 + ... + x_m)/m$$

La *desviación respecto a la media* es

$$d_i = (x_i - x_{media}),\ i = 1,2,\ ...,\ m$$

y la *desviación estándar* es

$$s = [(d_1^2 + d_2^2 + ... + d_m^2)/m]^{1/2}$$

Utilizar el programa para procesar el siguiente conjunto de datos:

27.5	87.0
13.4	39.9
53.8	47.7
29.2	8.1
74.5	63.2

Repetir el cálculo para k listas diferentes de números. Calcular la media global, la desviación estándar global, el máximo absoluto (mayor) y el mínimo absoluto (menor algebraico).

d) Suponer que tenemos un conjunto de valores tabulados de x e y, esto es,

y_0	y_1	y_2	...	y_n
x_0	x_1	x_2	...	x_n

y queremos obtener el valor de y para algún valor de x que cae entre dos de los valores tabulados. Este problema se resuelve normalmente mediante *interpolación*, pasando un polinomio $y(x)$ a través de n puntos tales que $y(x_0) = y_0$, $y(x_1) = y_1$, ..., $y(x_n) = y_n$ y después evaluando y para el valor deseado de x.

Una forma común de realizar la interpolación es usar la *fórmula de Lagrange* de la interpolación polinomial. Para hacer esto se escribe

$$y(x) = f_0(x)y_0 + f_1(x)y_1 + \ .\ .\ .\ + f_n(x)y_n$$

donde $f_i(x)$ es un polinomio tal que

$$f_i(x) = \left[\frac{(x-x_0)(x-x_1)...(x-x_{i-1})(x-x_{i+1})...(x-x_n)}{(x_i-x_0)(x_i-x_1)...(x_i-x_{i-1})(x_i-x_{i+1})...(x_i-x_n)}\right]$$

Notar que $f_i(x_i) = 1$ y $f_i(x_j) = 0$, donde x_j es un valor tabulado de x distinto de x_i. Por tanto, se asegura que $y(x_i) = y_i$.

Escribir un programa en C que lea n pares de datos, donde n no excede de 10, y obtenga a continuación un valor interpolado de y para uno o más valores especificados de x. Usar el programa para obtener valores interpolados de y en $x = 13.7$, $x = 37.2$, $x = 112$ y $x = 147$ a partir de los datos dados a continuación. Determinar cuántos pares tabulados de datos se requieren en cada cálculo para obtener un valor razonablemente ajustado de y.

y = 0.21073	x = 0
0.45482	20
0.49011	30
0.50563	40
0.49245	50
0.47220	60
0.43433	80
0.33284	120
0.19390	180

9.48. Los siguientes problemas están relacionados con juegos de azar (juegos de apuestas). Cada problema requiere el uso de números aleatorios, como se describe en el Ejemplo 7.11. Cada programa requiere también el uso de un array. Los programas deben ser interactivos y estar modularizados.

a) Escribir un programa en C que simule un juego de «blackjack» entre dos jugadores. La computadora no será un participante en el juego, simplemente dará las cartas a cada jugador y proveerá a cada jugador con una o más cartas adicionales cuando éste lo solicite.

Las cartas se dan en orden, primero una carta a cada jugador, después otra carta a cada uno. Se pueden demandar cartas adicionales.

El objeto del juego es obtener 21 puntos, o tantos puntos como sea posible sin exceder de 21 en cada mano. Un jugador es automáticamente descalificado si las cartas en su mano exceden de 21 puntos. Las figuras cuentan 10 puntos y un as puede contar un punto u 11 puntos. Así, un jugador puede obtener 21 puntos («¡blackjack!») si tiene un as y una figura o un 10. Si el jugador tiene menos puntos con sus dos primeras cartas, puede pedir una carta o más, mientras su puntuación no pase de 21.

Utilizar números aleatorios para simular el reparto de las cartas. Asegurar la inclusión de una condición para que la misma carta no sea dada más de una vez.

b) A la *ruleta* se juega con una rueda que contiene 38 cuadros diferentes en su circunferencia. Dos de los cuadros, numerados con el 0 y 00, son verdes; 18 cuadros son rojos y 18 son negros. Se alternan los cuadros rojos y negros y están numerados de 1 a 36 en orden aleatorio.

Una pequeña bola gira dentro de la rueda, que como resultado termina quedando dentro de una ranura debajo de uno de los cuadros. El juego es apostar al resultado de los giros, de una de las maneras siguientes:

i) Seleccionando un cuadro rojo o negro, con una paridad de 35 a 1. Así, si el jugador apuesta 1 dólar y gana, recibirá un total de 36 dólares: el original más otros 35 dólares.
ii) Seleccionando un color, rojo o negro, con una paridad 1 a 1. Así, si el jugador elige rojo y apuesta 1 dólar, si la bola se para debajo de un cuadro rojo recibirá 2 dólares.
iii) Seleccionando los números pares o impares (excluidos 0 y 00), con paridad 1 a 1.
iv) Seleccionando los 18 números bajos o los 18 altos, con paridad 1 a 1.

El jugador perderá automáticamente si la bolita se para debajo de uno de los cuadros verdes (0 o 00).

Escribir un programa interactivo en C que simule el juego de la ruleta. Permitir que los jugadores seleccionen cualquier tipo de apuesta que deseen eligiéndola en un menú. Escribir el resultado de cada juego seguido por un mensaje apropiado que indique si el jugador ha ganado o ha perdido.

c) Escribir un programa interactivo en C que simule un juego de BINGO. Escribir cada combinación de letra-número según sea sacada (generada). Asegurarse que una combinación no se saca más de una vez. Recordar que cada una de las letras de B-I-N-G-O corresponde a un cierto rango de números, como se indica a continuación.

B: 1 – 15
I: 16 – 30
N: 31 – 45
G: 46 – 60
O: 61 – 75

Cada jugador tendrá un cartón con cinco columnas, rotuladas B-I-N-G-O. Cada columna contendrá cinco números, dentro de los rangos indicados arriba. Dos jugadores no pueden tener el mismo cartón. El primer jugador que tenga una línea de números sacados (en vertical, horizontal o en diagonal) gana.

Nota: A veces la posición central de cada cartón se cubre antes de comenzar el juego (una jugada «gratis»). A veces también se juega a que deben salir *todos* los números del cartón para ganar.

9.49. Escribir un programa interactivo en C que codifique y decodifique una línea de texto. Para codificar una línea de texto, proceder como sigue:

1. Convertir cada carácter, incluidos espacios en blanco, en su equivalente ASCII.
2. Generar un entero positivo aleatorio. Añadir este entero al ASCII equivalente a cada carácter. El mismo entero aleatorio será usado para la línea de texto completa.
3. Suponer que `N1` representa el menor valor permisible en ASCII y `N2` representa el mayor valor permisible en ASCII. Si el número obtenido en el paso 2 anterior (el ASCII original más el entero aleatorio) excede de `N2`, se le resta el múltiplo mayor posible de `N2` y se suma el resto a `N1`. Por tanto, el número codificado estará siempre entre `N1` y `N2`, y siempre representará algún carácter ASCII.
4. Escribir los caracteres correspondientes a los valores ASCII codificados.

El procedimiento se invierte cuando se decodifica una línea de texto. Asegurarse que se usa para decodificar el mismo entero que fue usado para codificar.

CAPÍTULO 10

Punteros

Un *puntero* es una variable que representa la *posición* (en vez del valor) de otro dato, tal como una variable o un elemento de un array. Los punteros son usados frecuentemente en C y tienen gran cantidad de aplicaciones. Por ejemplo, pueden ser usados para trasvasar información entre una función y sus puntos de llamada. En particular proporcionan una forma de devolver varios datos desde una función a través de los argumentos de la función. Los punteros también permiten que referencias a otras funciones puedan ser especificadas como argumentos de una función. Esto tiene el efecto de pasar funciones como argumentos de una función dada.

Los punteros están muy relacionados con los arrays y proporcionan una vía alternativa de acceso a los elementos individuales del array. Es más, proporcionan una forma conveniente para representar arrays multidimensionales, permitiendo que un array multidimensional sea reemplazado por un array de punteros de menor dimensión. Esta característica permite que una colección de cadenas de caracteres sean representadas por un solo array, incluso cuando las cadenas puedan tener distinta longitud.

10.1. CONCEPTOS BÁSICOS

Dentro de la memoria de la computadora, cada dato almacenado ocupa una o más celdas contiguas de memoria (es decir, palabras o bytes adyacentes). El número de celdas de memoria requeridas para almacenar un dato depende de su tipo. Por ejemplo, un carácter se almacenará normalmente en un byte (8 bits) de memoria; un entero usualmente necesita dos bytes contiguos; un número en coma flotante puede necesitar cuatro bytes contiguos; y una cantidad en doble precisión puede requerir ocho bytes contiguos. (Véanse Capítulo 2 y Apéndice D.)

Supongamos que `v` es una variable que representa un determinado dato. El compilador automáticamente asignará celdas de memoria para este dato. Podemos acceder al dato si conocemos la localización (*dirección*) de la primera celda de memoria *. La dirección de memoria de `v` puede ser determinada mediante la expresión `&v`, donde `&` es un operador unario, llamado el *operador dirección*, que proporciona la dirección del operando.

* Las celdas adyacentes de memoria dentro de la computadora están numeradas consecutivamente, desde el principio hasta el fin del área de memoria. El número asociado con cada celda de memoria es conocido como la *dirección* de la celda. La mayoría de las computadoras usan el sistema de numeración decimal para designar las direcciones de celdas de memoria consecutivas, aunque algunas computadoras usan el sistema de numeración octal (ver Apéndice A).

Ahora vamos a asignar la dirección de v a otra variable, pv. Así,

```
pv = &v
```

Esta nueva variable es un *puntero* a v, puesto que «apunta» a la posición de memoria donde se almacena v. Recordar, sin embargo, que pv representa la *dirección* de v y no su valor. De esta forma, pv es referida como una *variable apuntadora*. La relación entre v y pv está ilustrada en la Figura 10.1.



Figura 10.1. Relación entre pv **y** v **(donde** pv = &v **y** v = *pv**)**

El dato representado por v (es decir, el dato almacenado en las celdas de memoria de v) puede ser accedido mediante la expresión *pv, donde * es un operador unario, llamado el *operador indirección*, que opera sólo sobre una variable puntero. Por tanto, *pv y v representan el mismo dato (el contenido de las mismas celdas de memoria). Además, si escribimos pv = &v y u = *pv, entonces u y v representan el mismo valor; esto es, el valor de v se asigna indirectamente a u. (Se supone que u y v están declaradas como del mismo tipo de datos.)

EJEMPLO 10.1. Mostramos a continuación un programa sencillo que ilustra la relación entre dos variables enteras, sus correspondientes direcciones y sus punteros asociados.

```
#include <stdio.h>

main()
{
   int u = 3;
   int v;
   int *pu;      /* puntero a un entero */
   int *pv;      /* puntero a un entero */

   pu = &u;      /* asigna dirección de u a pu */
   v = *pu;      /* asigna valor de u a v */
   pv = &v;      /* asigna dirección de v a pv */

   printf("\nu=%d   &u=%X   pu=%X   *pu=%d", u, &u, pu, *pu);
   printf("\n\nv=%d   &v=%X   pv=%X   *pv=%d", v, &v, pv, *pv);
}
```

Observe que pu es un puntero a u, y pv un puntero a v. Por tanto, pu representa la dirección de u y pv la dirección de v. (En la próxima sección se tratará la declaración de punteros.)

La ejecución de este programa produce la siguiente salida:

```
u=3   &u=F8E   pu=F8E   *pu=3

v=3   &v=F8C   pv=F8C   *pv=3
```

En la primera línea vemos que u representa el valor 3, como está especificado en la instrucción de declaración. La dirección de u está directamente determinada por el compilador como F8E (hexadecimal). Al puntero pu se le asigna este valor; por tanto pu también representa la dirección F8E (hexadecimal). Finalmente el valor al que apunta pu (el valor almacenado en la celda de memoria cuya dirección es F8E) es 3, como era de esperar.

Análogamente, la segunda línea muestra que v representa el valor 3. Esto era lo esperado, ya que hemos asignado el valor *pu a v. La dirección de v, y por tanto el valor de pv, es F8C. Notar que u y v tienen direcciones diferentes. Y finalmente, vemos que el valor al que apunta pv es 3, como era de esperar.

La relación entre pu y u, y pv y v, es mostrada en la Figura 10.2. Notar que las posiciones de las variables punteros (direcciones EC7 para pu y EC5 para pv) no son escritas por el programa.

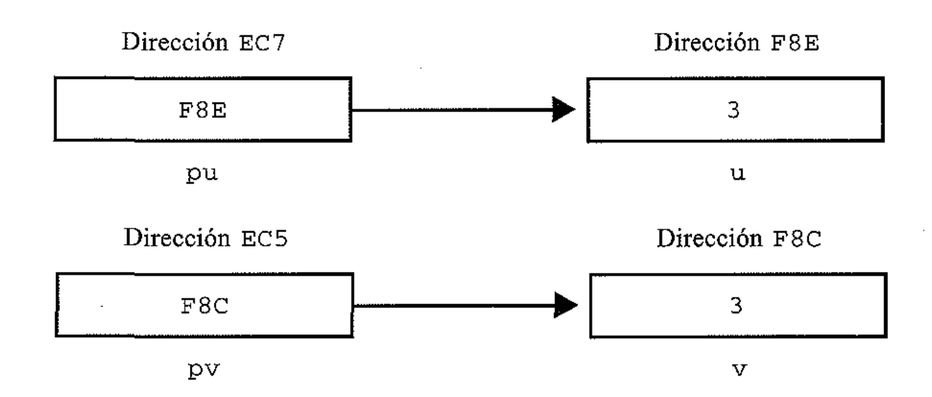


Figura 10.2.

Los operadores unarios & y * son miembros del mismo grupo de precedencia que los otros operadores unarios, -, ++, --, !, sizeof y (tipo), los cuales fueron presentados en el Capítulo 3. Recordar que este grupo de operadores tiene mayor precedencia que los grupos que contienen a los operadores aritméticos, y que la asociatividad de los operadores unarios es de derecha a izquierda (ver Apéndice C).

El operador dirección (&) sólo puede actuar sobre operandos con dirección única, como variables ordinarias o elementos individuales de un array. Por consiguiente, *el operador dirección no puede actuar sobre expresiones aritmética*, tales como 2 * (u + v).

El operador indirección (*) sólo puede actuar sobre operandos que sean punteros (por ejemplo, variables puntero). Sin embargo, si pv apunta a v (esto es, pv = &v), entonces una expresión como *pv puede intercambiarse con la correspondiente variable v. Así una referencia indirecta (por ejemplo, *pv) puede aparecer en lugar de una variable (por ejemplo v) dentro de una expresión más complicada.

EJEMPLO 10.2. Considerar el siguiente programa sencillo en C.

```
#include <stdio.h>

main()
{
```

```
    int u1, u2;
    int v = 3;
    int *pv;                  /* pv apunta a v */

    u1 = 2 * (v + 5);         /* expresión ordinaria */

    pv = &v;                  /* asigna dirección de v a pv */
    u2 = 2 * (*pv + 5);       /* expresión equivalente */

    printf("\nu1=%d   u2=%d", u1, u2);
}
```

Este programa involucra el uso de dos expresiones enteras. La primera, `2 * (v + 5)`, es una expresión aritmética ordinaria, mientras que la segunda, `2 * (*pv + 5)`, implica el uso de un puntero. Las expresiones son equivalentes, ya que `v` y `*pv` representan el mismo valor entero.

La siguiente salida es generada cuando se ejecuta el programa:

```
u1=16   u2=16
```

Una referencia indirecta puede aparecer también en la parte izquierda de una instrucción de asignación. Esto proporciona otro método para asignar un valor a una variable o a un elemento de un array.

EJEMPLO 10.3. A continuación se muestra un programa sencillo en C.

```
#include <stdio.h>

main()
{
    int v = 3;
    int *pv;

    pv = &v;                              /* pv apunta a v */
    printf("\n*pv=%d   v=%d", *pv, v);

    *pv = 0;                              /* reasigna v indirectamente */
    printf("\n\n*pv=%d   v=%d", *pv, v);
}
```

El programa empieza asignando un valor inicial de 3 a la variable entera `v` y asignando la dirección de `v` a la variable puntero `pv`. Así `pv` se convierte en un puntero a `v`. La expresión `*pv` representa por tanto el valor 3. La primera instrucción `printf` se usa para ilustrar esto al escribir los valores actuales de `*pv` y `v`.

A continuación de la primera instrucción `printf`, el valor de `*pv` es puesto a cero. Por tanto, `v` tendrá reasignado el valor `0`. Esto se ilustra mediante la segunda instrucción `printf`, que hace que se escriban los nuevos valores de `*pv` y `v`.

Cuando se ejecuta el programa, se genera la siguiente salida:

```
*pv=3     v=3
*pv=0     v=0
```

Así el valor de `v` ha sido modificado al asignar un nuevo valor a `*pv`.

Las variables puntero pueden apuntar a variables numéricas o de carácter, a arrays, a funciones o a otras variables puntero. (También pueden apuntar a otros tipos de estructuras de datos que veremos posteriormente en este libro.) Por tanto, a una variable puntero se le puede asignar la dirección de una variable ordinaria (por ejemplo, `pv = &v`). También se le puede asignar la dirección de otra variable puntero (por ejemplo, `pv = px`), siempre que ambas variables puntero apunten al mismo tipo de datos. Además, a una variable puntero se le puede asignar un valor *nulo* (cero), como se explica en la sección 10.2. Por otra parte, las variables *ordinarias no pueden* ser asignadas a direcciones arbitrarias (por ejemplo, una expresión como `&x` no puede aparecer en la parte izquierda de una instrucción de asignación).

La sección 10.5 presenta información adicional concerniente a las operaciones que pueden ser realizadas con punteros.

10.2. DECLARACIÓN DE PUNTEROS

Los punteros, como cualquier otra variable, deben ser declarados antes de ser usados dentro de un programa en C. Sin embargo, la interpretación de una declaración de puntero es un poco diferente de la declaración de otras variables. Cuando se declara una variable puntero, el nombre de la variable debe ir precedido por un asterisco (*). Éste identifica a la variable como un puntero. El tipo de datos que aparece en la declaración se refiere al *objeto* del puntero, esto es, el dato que se almacena en la dirección representada por el puntero, en vez del puntero mismo.

Así, una declaración de puntero puede ser escrita en términos generales como

```
tipo-dato *ptvar;
```

donde *ptvar* es el nombre de la variable puntero y *tipo-dato* se refiere al tipo de dato del objeto del puntero. Recordar que un asterisco debe preceder a *ptvar*.

EJEMPLO 10.4. Un programa en C contiene las siguientes declaraciones:

```
float u, v;
float *pv;
```

La primera línea declara `u` y `v` como variables en coma flotante. La segunda línea declara `pv` como una variable puntero cuyo objeto es una cantidad en coma flotante; es decir, `pv` apunta a una cantidad en coma flotante. Notar que `pv` representa una *dirección*, no una cantidad en coma flotante. (Algunas declaraciones adicionales de punteros son mostradas en los Ejemplos 10.1 a 10.3.)

Dentro de una declaración de variables se puede inicializar una variable puntero asignándole la dirección de otra variable. Recordar que la variable cuya dirección se asigna al puntero debe estar previamente declarada en el programa.

EJEMPLO 10.5. Un programa en C contiene las siguientes declaraciones:

```
float u, v;
float *pv = &v;
```

Las variables u y v son declaradas como variables en coma flotante y pv es declarada como una variable puntero que apunta a cantidades en coma flotante, como en el Ejemplo 10.4. Además, la dirección de v se asigna inicialmente a pv.

Esta terminología puede resultar confusa. Recordar que estas declaraciones son equivalentes a escribir

```
float u, v;        /* declaración de variables en coma flotante */
float *pv;         /* declaración de variable puntero */
. . . . .
pv = &v;           /* asignar la dirección de v a pv */
```

Notar que no se incluye un asterisco en la instrucción de asignación.

En general no tiene sentido asignar un valor entero a una variable puntero. Pero una excepción es la asignación de 0, que a veces se utiliza para indicar algunas condiciones especiales. En tales situaciones, la práctica recomendada es definir una constante simbólica NULL que represente el valor 0 y usar NULL en la inicialización del puntero. Esta práctica enfatiza el hecho de que la asignación del cero representa una situación especial.

EJEMPLO 10.6. Un programa en C contiene las siguientes definiciones de constantes simbólicas y declaraciones:

```
#define NULL 0
float u, v;
float *pv = NULL;
```

Las variables u y v son declaradas como variables en coma flotante y pv es declarada como una variable puntero que apunta a cantidades en coma flotante. Además, pv tiene asignado inicialmente el valor 0 para indicar alguna condición especial dictada por la lógica del programa (que no se muestra en este ejemplo). El uso de la constante simbólica NULL sugiere que la asignación inicial es algo más que la asignación de un valor entero ordinario.

Veremos otros tipos de declaraciones de punteros posteriormente en este capítulo.

10.3. PASO DE PUNTEROS A UNA FUNCIÓN

A menudo los punteros son pasados a las funciones como argumentos. Esto permite que datos de la parte del programa en la que se llama a la función sean accedidos por la función, modificados dentro de ella y luego devueltos al programa de forma modificada. Referimos este uso de los punteros como pasar argumentos por *referencia* (o por *dirección* o por *posición*), en contraste con pasar argumentos por *valor*, como discutimos en el Capítulo 7.

Cuando se pasa un argumento por valor, el dato es *copiado* a la función. Así cualquier modificación hecha al dato dentro de la función no es devuelta a la rutina llamadora (ver sección 7.5). Sin embargo, cuando un argumento se pasa por referencia (por ejemplo cuando se pasa un puntero a una función) la *dirección* de dato es pasada a la función. El contenido de esta dirección puede ser accedido libremente, tanto dentro de la función como dentro de la rutina de llamada.

Además cualquier cambio que se realiza al dato (al contenido de la dirección) será reconocido en ambas, la función y la rutina de llamada. Así, el uso de punteros como argumentos de funciones permite que el dato sea modificado globalmente desde dentro de la función.

Cuando los punteros se utilizan como argumentos de una función, es necesario tener cuidado con la declaración de los argumentos formales dentro de la función. Los argumentos formales que sean punteros deben ir precedidos por un asterisco. Los prototipos de funciones se escriben de la misma manera. Si una declaración de función no incluye nombres de variables, el tipo de datos de cada argumento puntero debe ir seguido de un asterisco. El uso de argumentos puntero se ilustra en el siguiente ejemplo.

EJEMPLO 10.7. A continuación se muestra un programa sencillo en C que ilustra la diferencia entre argumentos ordinarios, que son pasados por valor, y argumentos puntero, que son pasados por referencia.

```
#include <stdio.h>

void func1(int u, int v);                /* prototipo de función */
void func2(int *pu, int *pv);            /* prototipo de función */

main()
{
   int u = 1;
   int v = 3;

   printf("\nAntes de la llamada a func1:   u=%d   v=%d", u, v);
   func1(u, v);
   printf("\nDespués de la llamada a func1: u=%d   v=%d", u, v);

   printf("\n\nAntes de la llamada a func2: u=%d   v=%d", u, v);
   func2(&u, &v);
   printf("\nDespués de la llamada a func2: u=%d   v=%d", u, v);
}

void func1(int u, int v)
{
   u = 0;
   v = 0;
   printf("\nDentro de func1:               u=%d   v=%d", u, v);
   return;
}

void func2(int *pu, int *pv)
{
   *pu = 0;
   *pv = 0;
   printf("\nDentro de func2:             *pu=%d *pv=%d", *pu, *pv);
   return;
}
```

Este programa contiene dos funciones, llamadas func1 y func2. La primera función, func1, recibe dos variables enteras como argumentos. Estas variables tienen originalmente asignados los valores 1 y 3, respectivamente. Los valores son modificados a 0 y 0 dentro de func1. Sin embargo, los nuevos valores no son reconocidos en main, porque los argumentos fueron pasados por valor y cualquier cambio sobre los argumentos es local a la función en la cual se han producido los cambios.

Consideremos ahora la segunda función, func2. Esta función recibe dos *punteros* a variables enteras como argumentos. Los argumentos son identificados como punteros por los operadores de indirección (los asteriscos) que aparecen en la declaración de los argumentos. Además, la declaración de argumentos indica que los punteros representan direcciones de cantidades *enteras*.

Dentro de func2 los contenidos de las direcciones apuntadas son reasignados con valores 0 y 0. Como las direcciones son reconocidas tanto en func2 como en main, los valores reasignados serán reconocidos dentro de main tras la llamada a func2. Por tanto, las variables enteras u y v habrán cambiado sus valores de 1 y 3 a 0 y 0.

Las seis instrucciones printf ilustran los valores de u y v y sus valores asociados *pu y *pv dentro de main y dentro de las dos funciones. Por tanto, cuando se ejecuta el programa se genera la siguiente salida:

```
Antes de la llamada a func1:   u=1    v=3
Dentro de func1:               u=0    v=0
Después de la llamada a func1: u=1    v=3

Antes de la llamada a func2:   u=1    v=3
Dentro de func2:             *pu=0  *pv=0
Después de la llamada a func2: u=0    v=0
```

Observe que los valores de u y v quedan sin modificar dentro de main después de la llamada a func1, mientras que los valores de esas variables cambian dentro de main después de la llamada a func2. Así la salida ilustra la naturaleza local de las modificaciones dentro de func1 y la global dentro de func2.

Este ejemplo contiene características adicionales que deben ser remarcadas. Observe, por ejemplo, el prototipo de función

```
void func2(int *pu, int *pv);
```

Los elementos entre paréntesis identifican los argumentos como punteros a cantidades enteras. Las variables puntero pu y pv no han sido declaradas en ninguna parte dentro de main. Esto está permitido en el prototipo de la función, ya que pu y pv son argumentos ficticios en vez de argumentos reales. También se podría haber escrito la declaración de la función sin los nombres de los argumentos como

```
void func2(int *, int *);
```

Consideremos ahora la declaración de los argumentos formales en la primera línea de la función func2, esto es,

```
void func2(int *pu, int *pv)
```

Los argumentos formales pu y pv son consistentes con los argumentos ficticios del prototipo de la función. En este ejemplo, los nombres de las variables correspondientes son los mismos, pero en general esto no es necesario.

Finalmente, notar la manera en que u y v son accedidos dentro de func2, esto es,

```
*pu = 0;
*pv = 0;
```

Así u y v son accedidos indirectamente, al referenciar el contenido de las direcciones representadas por los punteros pu y pv. Esto es necesario, ya que las variables u y v no se reconocen como tales dentro de func2.

Ya hemos mencionado el hecho de que un nombre de array es un puntero al array; esto es, el nombre del array representa la dirección del primer elemento del array (ver sección 9.3). Por tanto, un nombre de array se trata como un puntero cuando se pasa a una función. No obstante, no es necesario preceder el nombre del array con el símbolo & (ampersand) dentro de la llamada a la función.

Un nombre de array que aparece como argumento formal en una definición de función puede ser declarado como puntero o como array de tamaño no especificado, como se indica en la sección 9.3. La elección es cuestión de preferencia personal, pero a menudo vendrá determinada por la forma en la cual los elementos individuales del array sean accedidos dentro de la función (más sobre esto en la siguiente sección).

EJEMPLO 10.8. Análisis de una línea de texto. Supongamos que queremos analizar una línea de texto examinando cada carácter y determinando a qué categoría pertenece. En particular, supongamos que contamos el número de vocales, consonantes, dígitos, espacios en blanco y «otros» caracteres (puntuaciones, operadores, paréntesis, etc.). Esto puede realizarse fácilmente leyendo una línea de texto, almacenándola en un array unidimensional y analizando cada uno de sus elementos. Un contador apropiado será incrementado para cada carácter. El valor de cada contador (número de vocales, número de consonantes, etcétera) puede escribirse a continuación después de que todos los caracteres hayan sido analizados.

Escribamos un programa completo en C que efectúe dicho análisis. Para ello, primero definimos los siguientes identificadores.

linea	=	array de 80 caracteres que contendrá la línea de texto
vocales	=	contador entero que indica el número de vocales
consonantes	=	contador entero que indica el número de consonantes
digitos	=	contador entero que indica el número de dígitos
blancos	=	contador entero que indica el número de espacios en blanco (espacios en blanco o tabuladores)
otros	=	contador entero que indica el número de caracteres que no pertenecen a las anteriores categorías

Observe que los caracteres de nueva línea no son incluidos en la categoría «blancos», ya que no pueden estar dentro de una línea simple de texto.

Estructuraremos el programa de modo que la línea de texto se lea dentro de la parte principal del programa y sea pasada a la función donde será analizada. La función devolverá el valor de cada contador después de que todos los caracteres hayan sido analizados. Los resultados del análisis (el valor de cada contador) serán escritos desde la parte principal del programa.

El análisis real puede ser realizado con un bucle para examinar cada uno de los caracteres. Dentro del bucle primero se convierte cada carácter que sea una letra a mayúsculas. Esto evita la necesidad de distinguir entre letras mayúsculas y minúsculas. A continuación podemos clasificar el carácter utilizando instrucciones if - else anidadas. Una vez identificada la clase, se incrementa el correspondiente contador. El proceso completo es repetido hasta que se encuentra el carácter de fin de cadena (\0).

A continuación se muestra el programa completo en C.

```
/*  contar el número de vocales, consonantes, dígitos, espacios  en
    blanco y "otros" caracteres en una línea de texto */

#include <stdio.h>
#include <ctype.h>

/* prototipo de función */
void analiza_linea(char linea[], int *pv, int *pc, int *pd, int *pb, int *po);

main()
{
   char linea[80];             /* línea de texto */
   int vocales = 0;            /* contador de vocales */
   int consonantes = 0;        /* contador de consonantes */
   int digitos = 0;            /* contador de dígitos */
   int blancos = 0;            /* contador de espacios en blanco */
   int otros = 0;              /* contador del resto de caracteres */

   printf("Introducir una línea de texto:\n");
   scanf("%[^\n]", linea);

   analiza_linea(linea, &vocales, &consonantes, &digitos, &blancos,
              &otros);

   printf("\nNº de vocales: %d", vocales);
   printf("\nNº de consonantes: %d", consonantes);
   printf("\nNº de dígitos: %d", digitos);
   printf("\nNº de caracteres en blanco: %d", blancos);
   printf("\nNº de otros caracteres: %d", otros);
}

void analiza_linea(char linea[], int *pv, int *pc, int *pd, int *pb, int *po)

/* analizar los caracteres en una línea de texto */

{
   char c;                     /* carácter en mayúsculas */
   int cont = 0;               /* contador de caracteres */

   while ((c = toupper(linea[cont])) != '\0') {

       if (c == 'A' || c == 'E' || c == 'I' || c == 'O' || c == 'U')
          ++ *pv;                               /* vocal */
       else if (c >= 'A' && c <= 'Z')
              ++ *pc;                           /* consonante */
            else if (c >= '0' && c <= '9')
                   ++ *pd;                      /* dígito */
                 else if (c == ' ' || c == '\t')
                        ++ *pb;                 /* espacio en blanco */
                      else
                        ++ *po;                 /* otro */
       ++cont;
   }
   return;
}
```

Observe el prototipo de función para analiza_linea que aparece al principio del programa. En particular, notar el tipo de dato void y la manera en que se especifican los tipos de datos de los argumentos. Obsérvese la diferencia entre el argumento de array y el resto de los argumentos punteros.

Obsérvese también la manera en la que están escritos los argumentos reales en la llamada a analiza_linea. El argumento de array, linea, no va precedido por el operador &, ya que los arrays son punteros por definición. El operador & debe preceder a cada uno del resto de los argumentos, ya que se pasa a la función no su valor sino su dirección.

Consideremos ahora la función analiza_linea. Todos los argumentos formales, incluido linea, son punteros. No obstante, linea está definida como un array de tamaño no especificado, mientras que el resto de los argumentos están específicamente declarados como punteros. Es posible (y bastante normal) declarar linea como un puntero en vez de como un array. Así, la primera línea de analiza_linea puede ser escrita como

```
void analiza_linea(char *linea, int *pv, int *pc, int *pd, int *pb, int *po)
```

en vez de como está en el listado del programa. Para ser consistente, el correspondiente prototipo de función debe ser escrito de modo similar.

El incremento de los contadores también requiere cierta explicación. Primero, observe que el *contenido* de cada dirección (el *objeto* de cada puntero) es incrementado. Segundo, notar que cada expresión de indirección (por ejemplo, *pv) es *precedida* por el operador unario ++. Como los operadores unarios se evalúan de derecha a izquierda, nos aseguramos que se incremente el contenido de cada dirección y no la dirección misma.

A continuación se muestra un diálogo típico que puede ser obtenido cuando se ejecuta el programa. (La línea de texto introducida por el usuario está subrayada.)

```
Introducir una línea de texto:
Computadoras personales con mas de 1024 KB son ahora bastante comunes.
```

La salida correspondiente es:

```
Nº de vocales: 22
Nº de consonantes: 33
Nº de dígitos: 4
Nº de caracteres en blanco: 10
Nº de otros caracteres: 1
```

Así vemos que esta línea de texto contiene 22 vocales, 33 consonantes, cuatro dígitos, 10 caracteres en blanco (espacios en blanco) y uno de otros caracteres (el punto).

Recordar que la función scanf requiere que los argumentos que sean variables ordinarias vayan precedidos por ampersands (ver sección 4.4). No obstante, los nombres de array están exentos de este requerimiento. Esto pudo parecer algo misterioso en el Capítulo 4, pero ahora debe tener sentido, considerando lo que sabemos de nombres de array y direcciones. Así, la función scanf requiere que se especifique la *dirección* de los elementos que vayan a ser introducidos en la memoria de la computadora. Los ampersands (&) proporcionan un medio para acceder a las direcciones de las variables ordinarias unievaluadas. Los ampersands no son necesarios con nombres de arrays, ya que estos mismos nombres representan direcciones.

EJEMPLO 10.9. A continuación se muestra el esquema de la estructura de un programa en C (repetido del Ejemplo 4.5).

```
#include <stdio.h>

main()
{
  char concepto[20];
  int num_partida;
  float coste;
  . . . . .
  scanf("%s %d %f", concepto, &num_partida, &coste);
  . . . . .
}
```

La instrucción `scanf` hace que una cadena de caracteres, una cantidad entera y una en coma flotante sean introducidas en la computadora y almacenadas en las direcciones de memoria asociadas con `concepto`, `num_partida` y `coste`. Como `concepto` es el nombre de un array, está claro que representa una dirección. Por esto, `concepto` no necesita (no puede) ir precedido por un ampersand dentro de la instrucción `scanf`. Sin embargo, `num_partida` y `coste` son variables convencionales. Por tanto deben ser escritas como `&num_partida` y `&coste` dentro de la instrucción `scanf`. Se necesita el operador ampersand para acceder a la *dirección* de estas variables en lugar de su valor.

Si se utiliza la función `scanf` para introducir un solo elemento del array en vez de toda el array, el nombre del elemento del array debe ir precedido por un ampersand, como se muestra a continuación (del Ejemplo 9.8).

```
scanf("%f", &lista[cont]);
```

Se puede pasar una *parte* de un array, en vez de toda el array, a una función. Para hacerlo, la dirección del primer elemento a pasar ha de ser especificada como argumento. El resto del array, comenzando por el elemento especificado, será entonces pasado a la función.

EJEMPLO 10.10. A continuación se muestra el esquema de la estructura de un programa en C.

```
#include <stdio.h>

void procesar(float z[]);

main()
{
  float z[100];
  . . . . .
  /* introducir valores para los elementos de z */

  . . . . .

  procesar(&z[50]);

  . . . . .

}
```

```
void procesar(float f[])
{
   . . . . .

   /* procesar los elementos de f */

   . . . . .

   return;
}
```

z se declara dentro de main como un array de 100 elementos en coma flotante. Después de intro-ducir los elementos de z en la computadora, se pasa la dirección de z[50] (&z[50]) a la función procesar. Así los últimos 50 elementos de z (los elementos de z[50] al z[99]) estarán disponibles para procesar.

En la siguiente sección veremos que la dirección de z[50] se puede escribir como z + 50 en vez de &z[50]. Por tanto, la llamada a procesar puede hacerse con procesar(z + 50) en vez de procesar(&z[50]), como se ha hecho. Se puede utilizar cualquiera de los dos métodos, dependiendo de las preferencias del programador.

Dentro de procesar, el array correspondiente es referido como f. Este array se declara de coma flotante pero con tamaño no especificado. De esta manera, el que la función reciba sólo una parte de z es indiferente; si todos los elementos de z son modificados dentro de procesar, sólo los 50 últimos se verán afectados dentro de main.

Sería deseable declarar dentro de procesar el argumento formal f como un puntero a una cantidad en coma flotante en vez de como un nombre de array. Así, el esquema de procesar puede ser escrito como

```
void procesar(float *f)
{
   . . . . .
   /* procesar los elementos de f */
   . . . . .
   return;
}
```

Observe las diferencias entre la declaración de los argumentos formales en los dos esquemas de función. Ambas declaraciones son válidas.

Una función también puede devolver un puntero a la parte llamadora del programa. Para hacer esto, la definición de la función y cualquier declaración de la función debe indicar que la función devolverá un puntero. Esto se realiza precediendo el nombre de la función con un asterisco. El asterisco debe aparecer tanto en la definición de la función como en las declaraciones de la función.

EJEMPLO 10.11. A continuación se muestra el esquema de la estructura de un programa en C que transfiere un array de elementos de doble precisión a una función y devuelve un puntero a uno de los elementos del array.

```
#include <stdio.h>

double *analiza(double z[]);

main()
{
   double z[100];       /* declaración de array */
   double *pz;          /* declaración de array */

   /* introducir valores para elementos de z */

   . . . . .

   pz = analiza(z);

   . . . . .

}

double *analiza(double f[])
{
   double *pf;          /* declaración de puntero */

   . . . . .

   /* procesar los elementos de f */

   pf = . . . . .;

   return(pf);
}
```

Dentro de `main` vemos que `z` se declara como un array de `100` elementos de doble precisión y `pz` es un puntero a una cantidad en doble precisión. También vemos una declaración para la función `analiza`. Observe que `scanf` aceptará un array de elementos en doble precisión como argumento y devolverá un puntero (la dirección) a una cantidad en doble precisión. El asterisco precediendo el nombre de la función `(*analiza)` indica que la función devuelve un puntero.

Dentro de la definición de la función, la primera línea indica que `analiza` acepta un parámetro formal `(f[])` y devuelve un puntero a una cantidad en doble precisión. El parámetro formal será un array unidimensional de elementos en doble precisión. El esquema sugiere que la dirección de uno de los elementos del array se asigna a `pf` durante o después del procesamiento de los elementos del array. Esta dirección es devuelta a `main`, donde se asigna a la variable puntero `pz`.

10.4. PUNTEROS Y ARRAYS UNIDIMENSIONALES

Recordar que el nombre de un array es realmente un puntero al primer elemento de ese array. Por tanto, si `x` es un array unidimensional, entonces la dirección del primer elemento del array se puede expresar tanto como `&x[0]` o simplemente como `x`. Además, la dirección del segundo

elemento del array se puede escribir tanto como `&x[1]` o como `(x + 1)`, y así sucesivamente. En general, la dirección del elemento `(i + 1)` del array se puede expresar bien como `&x[i]` o como `(x + i)`. Por tanto, tenemos dos métodos diferentes de escribir la dirección de cualquier elemento del array: podemos escribir el elemento real del array precedido por un ampersand (`&`); o podemos escribir una expresión en la cual el índice se añade al nombre del array.

En el último caso, se debe entender que estamos tratando con un tipo muy especial e inusual de expresión. En la expresión `(x + i)`, por ejemplo, `x` representa una dirección, mientras que `i` representa una cantidad entera. Además, `x` es el nombre de un array cuyos elementos pueden ser caracteres, enteros, cantidades en coma flotante, etc. (aunque todos los elementos del array han de ser del mismo tipo de datos). Por tanto, no estamos simplemente añadiendo valores numéricos. Más bien estamos especificando una dirección que está un cierto número de posiciones de memoria más allá del primer elemento del array. En términos más simples, estamos especificando una localización que está `i` elementos del array más allá del primero. Por tanto, la expresión `(x + i)` es una representación simbólica de una especificación de dirección en vez de una expresión aritmética.

Recordar que el número de celdas de memoria asociadas con un elemento del array depende del tipo de datos del array, así como de la propia arquitectura de la computadora. Por ejemplo, con algunas computadoras una cantidad entera ocupa dos bytes (dos celdas de memoria), una cantidad en coma flotante requiere cuatro bytes y una cantidad en doble precisión necesita ocho bytes. En otras computadoras, una cantidad entera puede ocupar cuatro bytes, y las cantidades en coma flotante y en doble precisión pueden requerir ocho bytes cada una. Y así sucesivamente.

Sin embargo, cuando se escribe la dirección de un elemento del array de la forma `(x + i)`, el programador no tiene que preocuparse por el número de celdas de memoria asociadas con cada tipo de datos del array; el compilador de C lo ajusta automáticamente. El programador debe indicar sólo la dirección del primer elemento (el nombre del array) y el número de elementos del array más allá del primero (valor del índice). El valor de `i` se denomina a veces *desplazamiento* (en inglés, *offset*) cuando se usa de esta manera.

Puesto que `&x[i]` y `(x + i)` representan la dirección del *i*-ésimo elemento de `x`, parece razonable que `x[i]` y `*(x + i)` representen el contenido de esa dirección, es decir, el *valor* del elemento *i*-ésimo de `x`. Esto es realmente cierto. Los dos términos son intercambiables. Cada término se puede usar en cualquier aplicación particular. La elección depende de las preferencias individuales del programador.

EJEMPLO 10.12. A continuación se presenta un programa que ilustra la relación entre los elementos de un array y sus direcciones.

```
#include <stdio.h>

main()
{
   static int x[10] = {10, 11, 12, 13, 14, 15, 16, 17, 18, 19};
   int i;
```

```
    for (i = 0; i <= 9; ++i) {
        /* escribir un elemento del array */
        printf("\ni= %d     x[i]= %d     *(x+i)= %d", i, x[i], *(x+i));

        /* escribir la correspondiente dirección del array */
        printf("     &x[i]= %X      x+i= %X", &x[i], (x+i));
    }
}
```

Este programa define un array x unidimensional de 10 elementos enteros, que tienen asignados valores 10, 11, ..., 19. La parte de acción del programa consiste en un bucle que muestra el valor y la dirección correspondiente para cada elemento del array. Observe que el valor de cada elemento del array se especifica de dos formas distintas, como x[i] y como *(x+i) para demostrar su equivalencia. Análogamente, la dirección de cada elemento del array está expresada en dos formas diferentes, como &x[i] y como (x+i), por la misma razón. Por tanto, el valor y la dirección de cada elemento del array aparecerá dos veces.

La ejecución del programa genera la siguiente salida:

```
i= 0     x[i]= 10     *(x+i)= 10     &x[i]= 72     x+i= 72
i= 1     x[i]= 11     *(x+i)= 11     &x[i]= 74     x+i= 74
i= 2     x[i]= 12     *(x+i)= 12     &x[i]= 76     x+i= 76
i= 3     x[i]= 13     *(x+i)= 13     &x[i]= 78     x+i= 78
i= 4     x[i]= 14     *(x+i)= 14     &x[i]= 7A     x+i= 7A
i= 5     x[i]= 15     *(x+i)= 15     &x[i]= 7C     x+i= 7C
i= 6     x[i]= 16     *(x+i)= 16     &x[i]= 7E     x+i= 7E
i= 7     x[i]= 17     *(x+i)= 17     &x[i]= 80     x+i= 80
i= 8     x[i]= 18     *(x+i)= 18     &x[i]= 82     x+i= 82
i= 9     x[i]= 19     *(x+i)= 19     &x[i]= 82     x+i= 84
```

La salida ilustra claramente la distinción entre x[i], que representa el valor del *i*-ésimo elemento del array, y &x[i], que representa su dirección. Además, vemos que el valor del *i*-ésimo elemento se puede representar por x[i] o *(x+i), y su dirección por &x[i] o x+i. Así observamos otra analogía entre *(x+i), que también representa el valor del *i*-ésimo elemento, y x+i, que también representa su dirección.

En particular observe que al primer elemento del array (correspondiente a i = 0) le ha sido asignado el valor 10 y una dirección 72 (hexadecimal). El segundo elemento tiene el valor 11 y dirección 74, etc. Así, la posición de memoria 72 contendrá el valor entero 10, la 74 contendrá el 11, etc.

Debe quedar claro que el compilador asigna automáticamente estas direcciones.

Cuando asignamos un *valor* a un elemento del array tal como x[i], la parte izquierda de la asignación puede ser escrita como x[i] o como *(x + i). De esta manera, un valor puede ser asignado directamente a un elemento del array o al área de memoria cuya dirección es la del elemento del array.

Por otra parte, a veces es necesario asignar una *dirección* a un identificador. En tales situaciones una variable puntero debe aparecer en la parte izquierda de la asignación. *No* es posible asignar una dirección arbitraria a un nombre de array o a un elemento del mismo. Por tanto, expresiones tales como x, (x + i) y &x[i] no pueden aparecer en la parte izquierda de una instrucción de asignación. Además, la dirección de un array no puede ser modificada arbitrariamente, por lo que expresiones como ++x no están permitidas.

EJEMPLO 10.13. Consideremos el esquema de la estructura del programa C mostrado a continuación.

```
#include <stdio.h>

main()
{
   int linea[80];
   int *pl;

   . . . . .

   /* asignar valores */
   linea[2] = linea[1];
   linea[2] = *(linea + 1);
   *(linea + 2) = linea[1];
   *(linea + 2) = *(linea + 1);

   /* asignar direcciones */
   pl = &linea[1];
   pl = linea + 1;
}
```

Cada una de las cuatro primeras instrucciones de asignación le asigna el valor del segundo elemento del array (`linea[1]`) al tercer elemento del array (`linea[2]`). Por tanto, las cuatro instrucciones son equivalentes. Un programador experimentado elegiría probablemente la primera o la cuarta, de modo que la notación fuera consistente.

Cada una de las dos últimas instrucciones de asignación asigna la *dirección* del segundo elemento del array al puntero `pl`. Podemos elegir hacer esto en un programa real si necesitamos «marcar» la dirección de `linea[1]` por alguna razón.

Observe que *la dirección de un elemento del array no puede ser asignada a otro elemento del array. No podemos* escribir una instrucción tal como

```
&linea[2] = &linea[1];
```

Por otra parte, si se desea podemos asignar el *valor* de un elemento del array a otro mediante un puntero, por ejemplo,

```
pl = &linea[1];
linea[2] = *pl;
```

o

```
pl = linea + 1;
*(linea + 2) = *pl;
```

Si un array numérico se define como una variable puntero, no se le pueden asignar valores iniciales a los elementos del array. Por tanto, será necesaria una definición de array convencional si se quiere asignar valores iniciales a los elementos de un array numérico. Sin embargo, a

una variable puntero *de tipo carácter* se le puede asignar una cadena de caracteres completa como parte de la declaración de la variable. De este modo, una cadena de caracteres puede ser convenientemente representada tanto por un array unidimensional de caracteres como por un puntero a carácter.

EJEMPLO 10.14. A continuación se muestra un programa sencillo en C en el que se representan dos cadenas de caracteres mediante arrays de caracteres unidimensionales.

```
#include <stdio.h>

char x[] = "Esta cadena se declara externamente\n\n";

main()
{
  static char y[] = "Esta cadena se declara dentro de main";
  printf("%s", x);
  printf("%s", y);
}
```

La primera cadena de caracteres está asignada al array externo `x[]`. La segunda cadena de caracteres está asignada al array estático `y[]`, que se define dentro de `main`. Esta segunda definición ocurre dentro de una función; por tanto `y[]` debe definirse como un array `static` para que pueda ser inicializado.

He aquí una versión diferente del mismo programa. Las cadenas de caracteres están ahora asignadas a variables puntero en vez de a arrays unidimensionales.

```
#include <stdio.h>

char *x = "Esta cadena se declara externamente\n\n";

main()
{
  char *y = "Esta cadena se declara dentro de main";

  printf("%s", x);
  printf("%s", y);
}
```

La variable puntero externa `x` apunta al comienzo de la primera cadena de caracteres, mientras que la variable puntero `y`, declarada dentro de `main`, apunta al comienzo de la segunda cadena de caracteres. Observe que `y` ahora puede inicializarse sin ser declarada como `static`.

La ejecución de cualquiera de los programas genera la siguiente salida:

```
Esta cadena se declara externamente

Esta cadena se declara dentro de main
```

Es posible sintácticamente, por supuesto, declarar una variable puntero `static`. Sin embargo, no hay razón para hacerlo en este ejemplo.

10.5. ASIGNACIÓN DINÁMICA DE MEMORIA

Puesto que un nombre de array es realmente un puntero al primer elemento dentro del array, debe ser posible definir el array como una variable puntero en vez de como un array convencional. Sintácticamente las dos definiciones son equivalentes. Sin embargo, la definición convencional de un array produce la reserva de un bloque fijo de memoria al principio de la ejecución del programa, mientras que esto no ocurre si el array se representa en términos de una variable puntero. Por consiguiente, el uso de una variable puntero para representar un array requiere algún tipo de asignación inicial de memoria, antes de que los elementos del array sean procesados. Esto se conoce como *asignación dinámica de memoria*. Generalmente, la función de biblioteca `malloc` se utiliza para este propósito, como se ilustra en el siguiente ejemplo.

EJEMPLO 10.15. Supongamos que `x` es un array unidimensional de `10` elementos enteros. Es posible definir `x` como una variable puntero en vez de como un array. Así, podemos escribir

```
int *x;
```

en vez de

```
int x[10];
```

o

```
#define TAM 10
int x[TAM];
```

Sin embargo, `x` no tiene asignado automáticamente un bloque de memoria cuando se define como una variable puntero, pero sí se le reservará por adelantado un bloque suficientemente grande para almacenar `10` enteros cuando `x` se define como un array.

Podemos usar la función de biblioteca `malloc` para asignar suficiente memoria para `x`, como sigue:

```
x = (int *) malloc(10 * sizeof(int));
```

Esta función reserva un bloque de memoria cuyo tamaño (en bytes) es equivalente a `10` cantidades enteras. Tal como está escrita, la función devuelve un puntero a un entero. Este puntero indica el comienzo del bloque de memoria. En general la conversión de tipo (en inglés «cast») que precede a `malloc` debe ser consistente con el tipo de datos de la variable puntero. Así, si `y` se hubiera definido como un puntero a una cantidad en doble precisión y solicitáramos la memoria suficiente para almacenar `10` cantidades en doble precisión, escribiríamos

```
y = (double *) malloc(10 * sizeof(double));
```

Sin embargo, si la declaración incluye la asignación de valores iniciales, entonces `x` *debe ser definido como un array en vez de como una variable puntero.* Por ejemplo:

```
int x[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

o

```
int x[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

Cuando se programa en C es usual utilizar expresiones con punteros en vez de referencias a elementos individuales del array. El programa resultante puede aparecer extraño al principio, pero deja de serlo al acostumbrarnos a acceder a valores almacenados en direcciones específicas. Generalmente sólo se necesita un poco de práctica.

EJEMPLO 10.16. Reordenación de una lista de números. Para ilustrar el uso de punteros con asignación dinámica de memoria, consideremos de nuevo el problema de reordenar una lista de enteros, como se describía en el Ejemplo 9.13. Pero ahora utilizaremos expresiones de punteros para acceder a valores individuales en vez de referirnos explícitamente a elementos individuales del array. En los otros aspectos presentamos un programa idéntico al dado en el Ejemplo 9.13.

Aquí está el programa completo en C.

```
/* reordenar un array unidimensional de enteros, de menor a mayor,
   usando notación de punteros */

#include <stdio.h>
#include <stdlib.h>

void reordenar(int n, int *x);

main()
{
   int i, n, *x;

   /* leer el valor de n */
   printf("\n¿Cuantos números serán introducidos? ");
   scanf("%d", &n);
   printf("\n");

   /* reserva de memoria */
   x = (int *) malloc(n * sizeof(int));

   /* leer la lista de números */
   for (i = 0; i < n; ++i) {
       printf("i = %d   x = ", i + 1);
       scanf("%d", x + i);
}

/* reordenar todos los elementos del array */
reordenar(n, x);

/* escribir la lista reordenada de números */
printf("\n\nLista de números reordenada:\n\n");
for (i = 0; i < n; ++i)
    printf("i = %d   x = %d\n", i + 1, *(x + i));
}

void reordenar(int n, int *x)    /* reordenar la lista de números */

{
```

```
    int i, elem, temp;

    for (elem = 0; elem < n - 1; ++elem)

        /* encontrar el menor del resto de los elementos */
        for (i = elem + 1; i < n; ++i)

            if (*(x + i) < *(x + elem)) {

                /* intercambiar los dos elementos */
                temp = *(x + elem);
                *(x + elem) = *(x + i);
                *(x + i) = temp;
            }
    return;
}
```

En este programa el array de enteros está definido como un puntero a un entero. La asignación inicial de memoria para la variable puntero se hace mediante la función de biblioteca `malloc`. En todo el programa se utiliza la notación de punteros para procesar los elementos individuales del array. Por ejemplo, el prototipo de función especifica ahora que el segundo argumento es un puntero a una cantidad entera en vez de un array de enteros. Este puntero identificará el comienzo del array de enteros.

También vemos que la función `scanf` especifica ahora la dirección del *i*-ésimo elemento como `x + i` en vez de `&x[i]`. Análogamente, la función `printf` representa ahora el valor del *i*-ésimo elemento como `*(x + i)` en vez de `x[i]`. Sin embargo, la llamada a `reordenar` es la misma que en el programa anterior, `reordenar(n, x);`.

Dentro de la función `reordenar` vemos que el segundo argumento formal se define ahora como una variable puntero en vez de como un array de enteros. Esto es consistente con el prototipo de función. Las diferencias más pronunciadas se encuentran en la instrucción `if`. En particular, observe que cada referencia a un elemento del array se escribe ahora como el contenido de una dirección. Así `x[i]` se escribe ahora como `*(x + i)`, y `x[elem]` como `*(x + elem)`. Esta instrucción `if` compuesta puede ser vista como un intercambio condicional entre los contenidos de dos direcciones, en vez de como un intercambio entre dos elementos diferentes de un array convencional.

Deberíamos comparar este programa con el mostrado en el Ejemplo 9.13 para apreciar las diferencias. Ambos programas generan los mismos resultados para los mismos datos de entrada. Sin embargo, el lector debería entender las diferencias sintácticas entre los dos programas.

Una ventaja importante de la asignación dinámica de la memoria es la posibilidad de reservar la memoria que se necesita durante la ejecución del programa y luego liberar esta memoria cuando no se necesita. Más aún, este proceso se puede repetir varias veces durante la ejecución del programa. Las funciones de biblioteca `malloc` y `free` se utilizan para estos propósitos, como se ilustra en el Ejemplo 11.32 (ver sección 11.6).

10.6. OPERACIONES CON PUNTEROS

En la sección 10.4 hemos visto que se puede sumar un valor entero a un nombre de array para acceder a un elemento individual del array. El valor entero se interpreta como el índice del array;

representa la posición relativa del elemento deseado con respecto al primer elemento del array. Esto funciona, ya que todos los elementos del array son del mismo tipo y por tanto cada elemento ocupa el mismo número de celdas de memoria (mismo número de bytes o palabras). El número de celdas que separan a dos elementos del array dependerá del tipo de datos del array, pero de esto se encarga el compilador automáticamente y por tanto no concierne al programador directamente.

Este concepto se puede ampliar a las variables puntero en general. De este modo, a una variable puntero se le puede sumar o restar un valor entero, pero el resultado de la operación debe ser interpretado con cuidado. Supongamos, por ejemplo, que `px` es una variable puntero que representa la dirección de una variable `x`. Podemos escribir expresiones tales como `++px`, `--px`, `(px + 3)`, `(px + i)` y `(px - i)`, donde `i` es una variable entera. Cada expresión representará una dirección localizada a cierta distancia de la posición original representada por `px`. La distancia exacta será el producto de la cantidad entera por el número de bytes o palabras que ocupa cada elemento al cual apunta `px`. Por ejemplo, supongamos que `px` apunta a un entero, y cada entero requiere dos bytes de memoria. Entonces la expresión `(px + 3)` dará como resultado una dirección que está seis bytes más allá del entero apuntado por `px`, como se ilustra en la Figura 10.3. Sin embargo, debe quedar claro que esta nueva dirección *no* tiene necesariamente que representar la dirección de otro elemento, en particular cuando los elementos almacenados entre las dos posiciones son de tipos de datos diferentes.

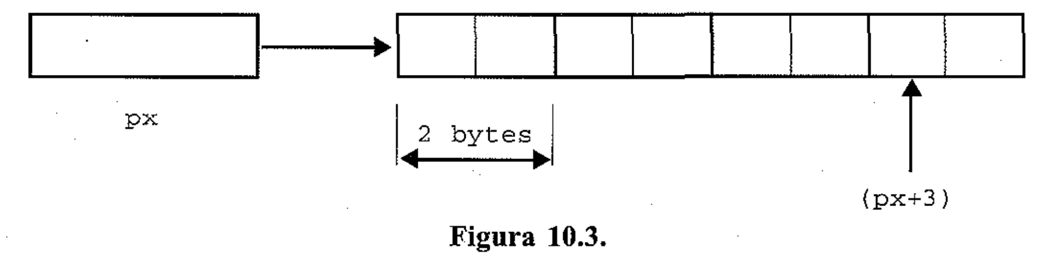


Figura 10.3.

EJEMPLO 10.17. Consideremos el programa en C que se muestra a continuación.

```
#include <stdio.h>

main()

{
  int *px;              /* puntero a un entero */
  int i = 1;
  float f = 0.3;
  double d = 0.005;
  char c = '*';

  px = &i;
  printf("Valores:    i=%i    f=%f    d=%f    c=%c\n\n", i, f, d, c);
  printf("Direcciones:    &i=%X    &f=%X    &d=%X    &c=%X\n\n",
                                                      &i, &f, &d, &c);
  printf("Valores de punteros:    px=%X    px + 1=%X    px + 2=%X");
  printf ("px + 3=%X",  px, px+1,  px+2,  px + 3);
}
```

Este programa escribe los valores y direcciones asociados con cuatro tipos diferentes de variables: i, una variable entera; f, una variable en coma flotante; d, una variable en doble precisión; y c, una variable carácter. El programa hace uso también de una variable puntero, px, que representa la dirección de i. También se muestran los valores de px, px + 1, px + 2 y px + 3, para que puedan ser comparados con las direcciones de las diferentes variables.

La ejecución del programa produce la siguiente salida:

```
Valores:    i=1    f=0.300000    d=0.005000    c=*

Direcciones:    &i=117E    &f=1180    &d=1186    &c=118E

Valores de punteros:    px=117E    px + 1=1180    px + 2=1182    px + 3=1184
```

La primera línea muestra simplemente el valor de las variables y la segunda las direcciones asignadas por el compilador. Observe que el número de bytes asociado con cada tipo de dato es diferente. Así, el valor entero representado por i requiere dos bytes (específicamente, direcciones 117E y 117F). El valor en coma flotante representado por f tiene asignados seis bytes (direcciones 1180 a 1185), pero sólo son usados cuatro bytes (direcciones 1180 a 1183). (Los compiladores gestionan el espacio de memoria según sus propias reglas.) Para el valor en doble precisión d son necesarios ocho bytes (direcciones 1186 a 118D). Y finalmente, el carácter representado por c empieza en la dirección 118E. Sólo se requiere un byte para almacenar este carácter, aunque la salida no indica el número de bytes entre este carácter y el próximo dato.

Consideremos ahora la tercera línea de la salida, que contiene las direcciones representadas por las expresiones de puntero. Está claro que px representa la dirección de i (117E). Esto no produce sorpresa, ya que esta dirección ha sido asignada a px mediante la expresión px = &i. Sin embargo, px + 1 se desplaza dos bytes , a 1180, px + 2 se desplaza otros dos bytes, a 1182, y así sucesivamente. La razón es que px apunta a un entero, y una cantidad entera necesita dos bytes con este compilador de C particular. Como resultado, cuando se suman constantes enteras a px, estas constantes se interpretan como múltiplo de dos bytes.

Si px se define como un puntero a otro tipo diferente de dato (un carácter o una cantidad en coma flotante), entonces cualquier entero sumado o restado al puntero se interpretará de manera distinta. En particular, cada valor entero representará un número equivalente de bytes individuales si px apunta a un carácter, o un número de bytes múltiplo de cuatro si px apunta a una cantidad en coma flotante. Se recomienda al lector que verifique esto por sí mismo.

Una variable puntero puede ser restada de otra siempre que ambas variables apunten a elementos del mismo array. El valor resultante indica el número de palabras o bytes que separan los correspondientes elementos del array.

EJEMPLO 10.18. En el programa mostrado a continuación, dos variables puntero apuntan al primero y al último elemento de un array de enteros.

```
#include <stdio.h>

main()
{
   int *px, *py;    /* punteros a entero */
   static int a[6] = {1, 2, 3, 4, 5, 6};
```

```
    px = &a[0];
    py = &a[5];
    printf("px=%X    py=%X", px, py);
    printf("\n\npy - px=%X", py - px);
}
```

En particular, la variable puntero px apunta a a[0] y py a a[5]. La diferencia, py - px, debería ser 5, ya que a[5] es el quinto elemento después de a[0].

La ejecución del programa produce la siguiente salida:

```
px=52   py=5C
py - px=5
```

La primera línea indica que la dirección de a[0] es 52 y la dirección de a[5] es 5C. La diferencia entre estos números hexadecimales es 10 (en decimal). Así a[5] está almacenado en una dirección que está 10 bytes más allá de la dirección de a[0]. Como cada cantidad entera ocupa dos bytes, deberíamos esperar que la diferencia entre py y px fuera 10/2=5. La segunda línea de la salida confirma este valor.

Las variables puntero pueden ser comparadas siempre que sean del mismo tipo de datos. Tales comparaciones pueden ser útiles cuando ambas variables apunten a elementos de un mismo array. Las comparaciones pueden probar la igualdad o desigualdad. Además, una variable puntero puede ser comparada con cero (normalmente expresado como NULL cuando se usa de esta manera, como se explicó en la sección 10.2).

EJEMPLO 10.19. Supongamos que px y py son variables puntero que apuntan a elementos de un mismo array. A continuación se muestran varias expresiones lógicas que involucran a estas dos variables. Todas las expresiones son sintácticamente correctas.

```
(px < py)

(px >= py)

(px == py)

(px != py)

(px == NULL)
```

Estas expresiones se pueden utilizar de la misma manera que cualquier otra expresión lógica. Por ejemplo,

```
if (px < py)
   printf("px < py");
else
   printf("px >= py");
```

Expresiones tales como (px < py) indican si el elemento asociado con px está situado delante o no del elemento asociado con py (si el índice asociado con *px es menor o no que el asociado con *py).

Debe quedar claro que las operaciones tratadas previamente son las *únicas* que se pueden realizar con punteros. Estas operaciones permitidas se resumen a continuación.

1. A una variable puntero se le puede asignar la dirección de una variable ordinaria (`pv = &v`).
2. A una variable puntero se le puede asignar el valor de otra variable puntero (por ejemplo, `pv = px`), siempre que ambos punteros apunten al mismo tipo de datos.
3. A una variable puntero se le puede asignar un valor nulo (cero) (por ejemplo, `pv = NULL`, donde `NULL` es una constante simbólica que representa el valor `0`).
4. A una variable puntero se le puede sumar o restar una cantidad entera (por ejemplo, `pv + 3`, `++pv`, etc.).
5. Una variable puntero puede ser restada de otra con tal que ambas apunten a elementos del mismo array.
6. Dos variables puntero pueden ser comparadas siempre que ambas apunten a datos del mismo tipo.

No se permiten otras operaciones aritméticas con punteros. Así, una variable puntero no puede ser multiplicada por una constante; no se pueden sumar dos punteros; y así sucesivamente. Se le recuerda al lector que a una variable ordinaria no se le puede asignar una dirección arbitraria (una expresión como `&x` no puede aparecer en la parte izquierda de una asignación).

10.7. PUNTEROS Y ARRAYS MULTIDIMENSIONALES

Como un array unidimensional puede ser representado en términos de un puntero (el nombre del array) y de un desplazamiento (el índice), es razonable esperar que los arrays multidimensionales pueden ser representadas también con una notación equivalente de punteros. En efecto, éste es el caso. Por ejemplo, un array bidimensional es en realidad una colección de arrays unidimensionales. Por tanto, podemos definir un array bidimensional como un puntero a un grupo de arrays unidimensionales contiguos. De este modo, podemos escribir una declaración de array bidimensional como

```
tipo-dato (*ptvar)[expresión 2];
```

en vez de

```
tipo-dato array[expresión 1][expresión 2];
```

Este concepto puede generalizarse para arrays multidimensionales, esto es,

```
tipo-dato (*ptvar)[expresión 2][expresión 3] . . . [expresión n];
```

reemplaza a

```
tipo-dato array[expresión 1][expresión 2] . . . [expresión n];
```

En estas declaraciones *tipo-dato* es el tipo de datos del array, *ptvar* el nombre de la variable puntero, *array* el nombre del array correspondiente y *expresión 1*, *expresión 2*, . . ., *expresión n* expresiones enteras positivas que indican el máximo número de elementos del array asociados con cada índice.

Observe los paréntesis que rodean al nombre del array y el asterisco que lo precede en la versión de puntero de cada declaración. Estos paréntesis deben estar presentes. Sin ellos estaremos definiendo un array de punteros en vez de un puntero a un grupo de arrays, ya que estos símbolos particulares (los corchetes y el asterisco) se evalúan normalmente de derecha a izquierda. Veremos más sobre este tema en la siguiente sección.

EJEMPLO 10.20. Supongamos que `x` es un array bidimensional de enteros con `10` filas y `20` columnas. Podemos declarar `x` como

```
int (*x)[20];
```

en vez de

```
int x[10][20];
```

En la primera declaración, `x` se define como un puntero a un grupo de arrays unidimensionales consecutivos de `20` elementos enteros. Así `x` apunta al primero de los arrays de `20` elementos, que es en realidad la primera fila (fila `0`) del array bidimensional original. Análogamente, `(x + 1)` apunta al segundo array de `20` elementos, que es la segunda fila (fila `1`) del array bidimensional original, y así sucesivamente, como se ilustra en la Figura 10.4.

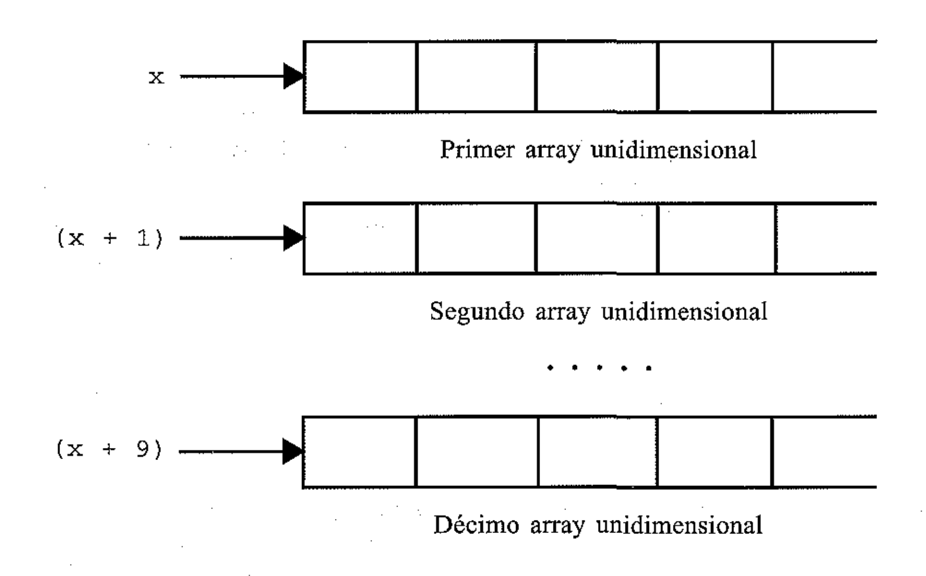


Figura 10.4.

Consideremos ahora el array tridimensional de números en coma flotante `t`. Este array se puede definir como

```
float (*t)[20][30];
```

en vez de

```
float t[10][20][30];
```

En la primera declaración, `t` se define como un puntero a un grupo de arrays bidimensionales consecutivas en coma flotante de `20 x 30`. Aquí `t` apunta al primer array de `20 x 30`, `(t + 1)` al segundo, y así sucesivamente.

Se puede acceder a un elemento individual de un array multidimensional mediante la utilización repetida del operador indirección. Sin embargo, normalmente este método es más difícil que el convencional para acceder a un elemento del array. El siguiente ejemplo ilustra el uso del operador indirección.

EJEMPLO 10.21. Supongamos que `x` es un array bidimensional de `10` filas y `20` columnas, como se declaró en el ejemplo anterior. El elemento en la fila `2`, columna `5`, puede ser accedido escribiendo

```
x[2][5]
```

o

```
*(*(x + 2) + 5)
```

La segunda forma requiere alguna explicación. Primero, notar que `(x + 2)` es un puntero a la fila `2`. Por tanto el objeto de este puntero, `*(x + 2)`, refiere toda la fila. Como la fila `2` es un array unidimensional, `*(x + 2)` es realmente un puntero al primer elemento de la fila `2`. Sumamos `5` a este puntero. Por tanto, `(*(x + 2) + 5)` es un puntero al elemento `5` (el sexto elemento) de la fila `2`. El objeto de este puntero, `*(*(x + 2) + 5)`, refiere al elemento en la columna `5` de la fila `2`, que es `x[2][5]`. La Figura 10.5 ilustra esta relación.

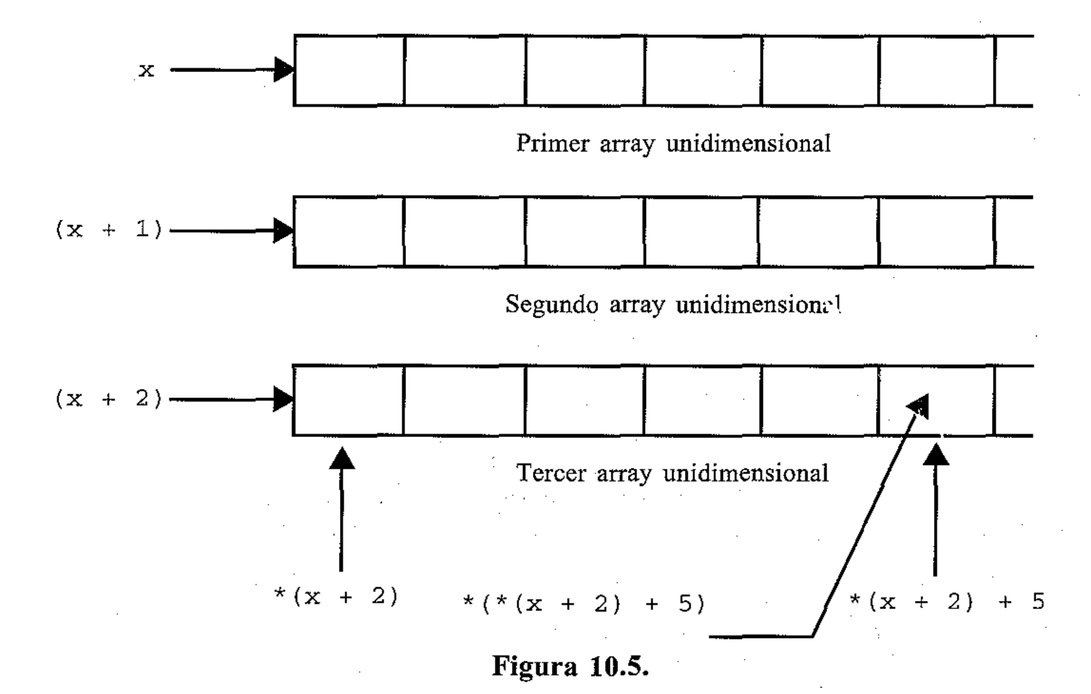


Figura 10.5.

Los programas que hacen uso de arrays multidimensionales pueden ser escritos de diversas maneras. En particular, hay diferentes maneras de definir los arrays, y diferentes formas de procesar los elementos individuales. La elección de un método u otro es un asunto de preferencias personales. En aplicaciones con arrays numéricos es a menudo más fácil definirlos del modo convencional, evitando así las posibles sutilezas asociadas con la asignación inicial de memoria. De todos modos, el siguiente ejemplo ilustra el uso de punteros para procesar arrays numéricos multidimensionales.

EJEMPLO 10.22. **Suma de dos tablas de números.** En el Ejemplo 9.19 desarrollamos un método para calcular la suma de los elementos correspondientes a dos tablas de enteros. El programa necesitaba tres arrays bidimensionales separados, que eran definidos de la forma convencional. Aquí tenemos una variación de ese programa, en que cada array bidimensional es definido como un puntero a un conjunto de arrays unidimensionales de enteros.

```
/* calcular la suma de los elementos en dos tablas de enteros */

/* cada array bidimensional se procesa como un puntero a un conjunto
   de arrays unidimensionales de enteros */

#include <stdio.h>
#include <stdlib.h>

#define MAXFIL 20

/* prototipos de funciones */
void leerentrada(int *a[MAXFIL], int nfilas, int ncols);
void calcularsuma(int *a[MAXFIL], int *b[MAXFIL],
                  int *c[MAXFIL], int nfilas, int ncols);
void escribirsalida(int *c[MAXFIL], int nfilas, int ncols);

main()
{
   int fila, nfilas, ncols;

   /* definiciones de punteros */
   int *a[MAXFIL], *b[MAXFIL], *c[MAXFIL];

   printf("¿Cuantas filas? ");
   scanf("%d", &nfilas);
   printf("¿Cuantas columnas? ");
   scanf("%d", &ncols);

   /* reserva inicial de memoria */
   for (fila = 0; fila < nfilas; ++fila) {
       a[fila] = (int *) malloc (ncols * sizeof(int));
       b[fila] = (int *) malloc (ncols * sizeof(int));
       c[fila] = (int *) malloc (ncols * sizeof(int));
   }

   printf("\n\nPrimera tabla:\n");
   leerentrada(a, nfilas, ncols);
```

```
   printf("\n\nSegunda tabla:\n");
   leerentrada(b, nfilas, ncols);

   calcularsuma(a, b, c, nfilas, ncols);

   printf("\n\nSumas de los elementos:\n\n");
   escribirsalida(c, nfilas, ncols);
}

void leerentrada(int *a[MAXFIL], int m, int n)
/* leer una tabla de enteros */
{

   int fila, col;

   for (fila = 0; fila < m; ++fila) {
       printf("\nIntroducir datos para la fila nº %2d\n", fila + 1);
       for (col = 0; col < n; ++col)
              scanf("%d", (*(a + fila) + col));
   }
   return;
}

void calcularsuma(int *a[MAXFIL], int *b[MAXFIL], int *c[MAXFIL],
                  int m, int n)
/* sumar los elementos de dos tablas de enteros */

{
   int fila, col;

   for (fila = 0; fila < m; ++fila)
       for (col = 0; col < n; ++col)
           *(*(c + fila) + col) = *(*(a + fila) + col) + *(*(b + fila)
                                                         + col);
   return;

}

void escribirsalida(int *a[MAXFIL], int m, int n)
/* escribir una tabla de enteros */

{
   int fila, col;

   for (fila = 0; fila < m; ++fila) {
       for (col = 0; col < n; ++col)
           printf("%4d", *(*(a + fila) + col));
       printf("\n");
   }
   return;
}
```

En este programa a, b y c son definidas como un array de punteros a enteros. Cada array tiene un máximo de MAXFIL elementos. Los prototipos de funciones y las declaraciones de argumentos formales dentro de las funciones subordinadas también representan los arrays de esta manera.

Puesto que cada elemento de a, b y c es un puntero, debemos reservar memoria inicial para cada fila de enteros, utilizando la función malloc, como se describe en la sección 10.5. Esta reserva de memoria aparece en main, después de haber sido introducidos los valores para nfilas y ncols. Consideremos la primera reserva de memoria; esto es,

```
a[fila] = (int *) malloc (ncols * sizeof(int));
```

En esta instrucción a[0] apunta a la primera fila. Análogamente, a[1] apunta a la segunda fila, a[2] a la tercera, y así sucesivamente. Por tanto, cada elemento del array apunta a un bloque de memoria suficientemente grande para almacenar una fila de enteros (ncols enteros). Reservas de memoria similares se han escrito para los otros dos arrays.

Los elementos individuales del array se procesan utilizando el operador indirección repetidamente. En leerentrada, por ejemplo, cada elemento del array es referenciado como

```
scanf("%d", (*(a + fila) + col));
```

De igual modo, la suma de los elementos del array dentro de calcularsuma se escribe como

```
*(*(c + fila) + col) = *(*(a + fila) + col) + *(*(b + fila) + col);
```

y la primera instrucción printf dentro de escribirsalida está escrita como

```
printf("%4d", *(*(a + fila) + col));
```

Por supuesto, podríamos haber utilizado la notación más convencional dentro de las funciones. Así, en leerentrada podríamos haber escrito

```
scanf("%d", 8 a[fila][col]);
```

en vez de

```
scanf("%d", (*(a + fila) + col));
```

Análogamente, en calcularsuma podríamos haber escrito

```
c[fila][col] = a[fila][col] + b[fila][col];
```

en vez de

```
*(*(c + fila) + col) = *(*(a + fila) + col) + *(*(b + fila) + col);
```

y en escribirsalida podríamos haber escrito

```
printf("%4d", a[fila][col]);
```

en vez de

```
printf("%4d", *(*(a + fila) + col));
```

Este programa generará idéntica salida que el mostrado en el Ejemplo 9.19 cuando se ejecuta con los mismos datos de entrada.

10.8. ARRAYS DE PUNTEROS

Un array multidimensional puede ser expresado como un array de punteros en vez de como un puntero a un grupo de arrays contiguos. En estos casos el nuevo array será de una dimensión menor que el array multidimensional. Cada puntero indicará el principio de un array de dimensión `(n-1)`.

En términos generales, un array bidimensional se puede definir como un array unidimensional de punteros escribiendo

```
tipo-dato *array[expresión 1];
```

en vez de la definición convencional del array,

```
tipo-dato array[expresión 1][expresión 2];
```

Análogamente, un array *n*-dimensional se puede definir como un array de punteros de dimensión `(n-1)` escribiendo

```
tipo-dato *array[expresión1][expresión2] . . . [expresiónn-1];
```

en vez de

```
tipo-dato array[expresión1][expresión2] . . . [expresiónn];
```

En estas declaraciones *`tipo-dato`* se refiere al tipo de datos del array *n*-dimensional original, *`array`* es el nombre del array y *`expresión 1, expresión 2`*, . . ., `expresión n` son expresiones enteras positivas que indican el máximo número de elementos asociados con cada índice.

Observe que el nombre del array precedido por un asterisco *no* está encerrado entre paréntesis en este tipo de declaración. (Comparar cuidadosamente con las declaraciones de punteros presentadas en la última sección.) Así la regla de precedencia de derecha a izquierda asocia el primer par de corchetes con `array`, definiéndolo como un array. El asterisco que lo precede establece que el array contendrá punteros.

Además, notar que la *última* expresión (extremo derecho) se omite cuando se define un array de punteros, mientras que la *primera* expresión (extremo izquierdo) se omite cuando se define un puntero a un grupo de arrays. (De nuevo comparar cuidadosamente con las declaraciones presentadas en la última sección.) El lector debe entender la distinción entre este tipo de declaración y la presentada en la sección anterior.

Cuando un array *n*-dimensional se expresa de esta manera, un elemento individual dentro del array *n*-dimensional puede ser accedido mediante un simple uso del operador indirección. El siguiente ejemplo ilustra cómo se hace esto.

EJEMPLO 10.23. Supongamos que x es un array bidimensional de enteros que tiene 10 filas y 20 columnas, como en el Ejemplo 10.20. Podemos definir x como un array unidimensional de punteros escribiendo

```
int *x[10];
```

Por tanto, x[0] apunta al primer elemento de la primera fila, x[1] al primer elemento de la segunda fila, y así sucesivamente. Observe que el número de elementos dentro de cada fila no está especificado explícitamente.

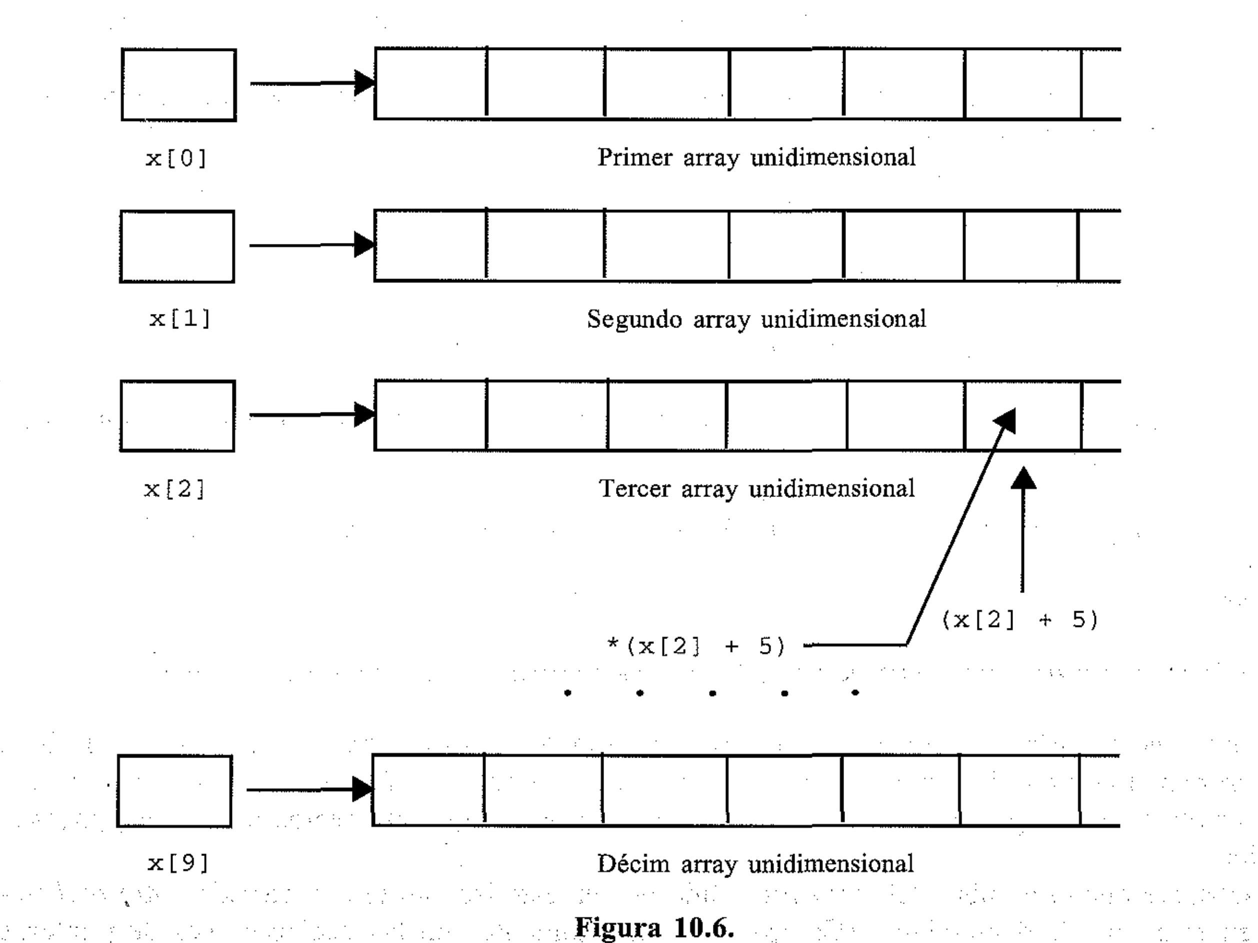


Figura 10.6.

Un elemento individual del array, tal como x[2][5], puede ser accedido escribiendo

```
*(x[2] + 5)
```

En esta expresión x[2] es un puntero al primer elemento en la fila 2, de modo que (x[2] + 5) apunta al elemento 5 (realmente el sexto elemento) de la fila 2. El objeto de este puntero, *(x[2] + 5), refiere por tanto a x[2][5]. Esta relación está ilustrada en la Figura 10.6.

Considerar ahora el array tridimensional en coma flotante `t`. Supongamos que la dimensionalidad de `t` es $10 \times 20 \times 30$. Este array puede ser expresado como un array bidimensional de punteros escribiendo

```
float *t[10][20];
```

Por tanto, tendremos `200` punteros (`10` filas, `20` columnas), cada uno apuntando a un array unidimensional.

Un elemento individual del array, tal como `t[2][3][5]`, puede ser accedido escribiendo

```
*(t[2][3] + 5)
```

En esta expresión `t[2][3]` es un puntero al primer elemento en el array unidimensional representado por `t[2][3]`. De aquí `(t[2][3] + 5)` apunta al elemento 5 (el sexto elemento) dentro de este array. El objeto de este puntero, `*(t[2][3] + 5)`, representa por tanto `t[2][3][5]`. Esta situación es, por supuesto, análoga al caso bidimensional descrito anteriormente.

EJEMPLO 10.24. Suma de dos tablas de números. Aquí tenemos otra versión del programa presentado en los Ejemplos 9.19 y 10.22, que calcula la suma de los elementos correspondientes de dos tablas de enteros. Ahora cada array bidimensional se representa como un array de punteros a arrays unidimensionales. Cada array unidimensional corresponderá a una fila del array bidimensional original.

```
/* calcular la suma de los elementos en dos tablas de enteros */

/* cada array bidimensional se procesa como un array de punteros

   cada puntero indica una fila del array bidimensional original */

#include <stdio.h>
#include <stdlib.h>

#define MAXFIL   20
#define MAXCOL   30

/* prototipos de funciones */
void leerentrada(int *a[MAXFIL], int nfilas, int ncols);
void calcularsuma(int *a[MAXFIL], int *b[MAXFIL],
                  int *c[MAXFIL], int nfilas, int ncols);
void escribirsalida(int *c[MAXFIL], int nfilas, int ncols);

main()
{
  int fila, nfilas, ncols;

  /* definiciones de arrays */
  int *a[MAXFIL], *b[MAXFIL], *c[MAXFIL];
```

```
    printf("¿Cuantas filas? ");
    scanf("%d", &nfilas);
    printf("¿Cuantas columnas? ");
    scanf("%d", &ncols);

    /* reserva inicial de memoria */
    for (fila = 0; fila <= nfilas; fila++) {
        a[fila] = (int *) malloc (ncols * sizeof(int));
        b[fila] = (int *) malloc (ncols * sizeof(int));
        c[fila] = (int *) malloc (ncols * sizeof(int));
    }

    printf("\n\nPrimera tabla: \n");
    leerentrada(a, nfilas, ncols);

    printf("\n\nSegunda tabla: \n");
    leerentrada(b, nfilas, ncols);

    calcularsuma(a, b, c, nfilas, ncols);

    printf("\n\nSumas de los elementos:\n\n");
    escribirsalida(c, nfilas, ncols);
}

void leerentrada(int *a[MAXFIL], int m, int n)
/* leer una tabla de enteros */
{
    int fila, col;

    for (fila = 0; fila < m; ++fila) {
        printf("\nIntroducir datos para la fila nº %2d\n", fila + 1);
        for (col = 0; col < n; ++col)
            scanf("%d", (a[fila] + col));
    }
    return;
}

void calcularsuma(int *a[MAXFIL], int *b[MAXFIL],
                  int *c[MAXFIL], int m, int n)
/* sumar los elementos de dos tablas de enteros */

{
    int fila, col;

    for (fila = 0; fila < m; ++fila)
        for (col = 0; col < n; ++col)
            *(c[fila] + col) = *(a[fila] + col) + *(b[fila] + col);
    return;
}

void escribirsalida(int *a[MAXFIL], int m, int n)
/* escribir una tabla de enteros */
```

```
{
   int fila, col;

   for (fila = 0; fila < m; ++fila) {
       for (col = 0; col < n; ++col)
           printf("%4d", *(a[fila] + col));
       printf("\n");
   }
   return;
}
```

Observe que a, b y c están definidos ahora como arrays unidimensionales de punteros. Cada uno contendrá MAXFIL elementos (MAXFIL punteros). Cada elemento del array apuntará a un array unidimensional de enteros. Los prototipos de funciones y las declaraciones de argumentos formales dentro de las funciones subordinadas también representan los arrays de esta manera.

Se debe reservar un bloque de memoria inicial para cada array unidimensional que es objeto de un puntero (cada fila dentro de cada una de las tablas). Esto lo realiza la función malloc. Por ejemplo, cada fila de la primera tabla tiene reservado espacio de memoria de la siguiente manera:

```
a[fila] = (int *) malloc (ncols * sizeof(int));
```

Esta instrucción asocia un bloque de memoria suficiente para almacenar ncols enteros con cada puntero (con cada elemento de a). Similar reserva de memoria se hace para b y c. Las instrucciones malloc se encuentran dentro de un for para reservar un bloque de memoria para cada una de las filas utilizadas dentro de las tres tablas.

Los elementos del array se procesan utilizando una combinación de notación de arrays y punteros. Por ejemplo, en leerentrada cada elemento del array es referenciado como

```
scanf("%d", (a[fila] + col));
```

Análogamente, en calcularsuma y escribirsalida los elementos del array se referencian como

```
*(c[fila] + col) = *(a[fila] + col) + *(b[fila] + col);
```

y

```
printf("%4d", *(a[fila] + col));
```

respectivamente. Estas instrucciones se pueden escribir con notación de array bidimensional si se desea.

Este programa, como el presentado en el Ejemplo 10.22, generará la misma salida que el mostrado en el Ejemplo 9.19 cuando se ejecuta con los mismos datos de entrada. Puede verificarlo por sí mismo. Sin embargo, si este programa fuera realizado desde el principio, el enfoque convencional mostrado en el Ejemplo 9.19, usando arrays bidimensionales, sería el escogido.

Los arrays de punteros ofrecen un método particularmente conveniente para almacenar cadenas de caracteres. En esta situación, cada elemento del array es un puntero de tipo carác-

ter que indica dónde comienza cada cadena. Así un array de n elementos puede apuntar a n cadenas diferentes. Cada cadena particular puede ser accedida refiriendo su puntero correspondiente.

EJEMPLO 10.25. Supongamos que las siguientes cadenas de caracteres tienen que ser almacenadas en un array de tipo carácter:

```
PACIFICO
ATLANTICO
INDICO
CARIBE
BERING
NEGRO
ROJO
NORTE
BALTICO
CASPIO
```

Estas cadenas se pueden almacenar en un array bidimensional de tipo carácter; por ejemplo,

```
char nombres[10][12];
```

Observe que `nombres` contiene `10` filas para las `10` cadenas. Cada fila debe ser suficientemente grande para almacenar por lo menos `10` caracteres, ya que `ATLANTICO` tiene nueve letras y el carácter nulo (`\0`) al final. Permitimos cada fila de hasta `12` caracteres, para tener la posibilidad de cadenas más largas.

Una forma mejor de hacer esto es definir un array de `10` punteros; esto es,

```
char *nombres[10];
```

De esta manera `nombres[0]` apuntará a `PACIFICO`, `nombres[1]` apuntará a `ATLANTICO`, y así sucesivamente. Observe que no es necesario incluir el máximo tamaño de cadena dentro de la declaración del array. Sin embargo, se tendrá que reservar una cantidad especificada de memoria para cada cadena de caracteres posteriormente en el programa, por ejemplo,

```
nombres[i] = (char *) malloc(12 * sizeof(char));
```

Una cadena de caracteres puede ser accedida refiriendo el correspondiente puntero (el correspondiente elemento del array); de este modo, un elemento de la cadena puede ser accedido mediante el uso del operador indirección. Por ejemplo, `*(*(nombres + 2) + 3)` es el cuarto carácter (el carácter número 3) en la tercera cadena (fila número 2) del array `nombres`, como se ha definido en el ejemplo precedente.

La reordenación de las cadenas de caracteres puede realizarse simplemente reasignando punteros (reasignando los elementos en un array de punteros). Las cadenas de caracteres no necesitan moverse.

EJEMPLO 10.26. **Reordenación de una lista de cadenas de caracteres.** Consideremos de nuevo el problema de introducir una lista de cadenas de caracteres en la computadora y reordenarlas alfabéticamen-

te. Vimos este problema en el Ejemplo 9.20, donde la lista se almacenaba en un array bidimensional. Trataremos este problema ahora utilizando un array unidimensional de punteros, donde cada puntero indica el principio de una cadena. El intercambio de cadenas puede realizarse ahora reasignando punteros cuando sea necesario.

A continuación se presenta el programa completo.

```
/*  ordenar alfabéticamente una lista de cadenas de caracteres,
    utilizando un array de punteros */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void reordenar(int n, char *x[]);

main()
{
   int i, n = 0;
   char *x[10];

   printf("Introducir debajo cada cadena en una línea\n\n");
   printf("Escribir \'FIN\' para terminar\n\n");

   /* leer la lista de cadenas de caracteres */
   do {
        /* reservar memoria */
        x[n] = (char *) malloc(12 * sizeof(char));

        printf("cadena %d: ", n + 1);
        scanf("%s", x[n]);
      }
   while (strcmp(x[n++], "FIN"));

   /* reordenar la lista de cadenas de caracteres */
   reordenar(--n, x);

   /* escribir la lista reordenada de cadenas de caracteres */
   printf("\n\nLista reordenada de cadenas:\n");
   for (i = 0; i < n; ++i)
       printf("\ncadena %d: %s", i + 1, x[i]);
}

void reordenar(int n, char *x[]) /* reordenar la lista de cadenas */

{
   char *temp;
   int i, elem;

   for (elem = 0; elem < n - 1; ++elem)
```

```
    /* encontrar la menor de las cadenas restantes */
    for (i = elem + 1; i < n; ++i)

         if (strcmp(x[elem], x[i]) > 0) {
            /* intercambiar las dos cadenas */
            temp = x[elem];
            x[elem] = x[i];
            x[i] = temp;
         }
    return;
}
```

La lógica es esencialmente la misma que la del Ejemplo 9.20, pero el array que contiene las cadenas de caracteres se define ahora como un array de punteros. Observe que el segundo argumento formal en la función `reordenar` se declara de la misma manera. Observar también la rutina de intercambio de cadenas (la instrucción `if`) dentro de `reordenar`. Ahora se intercambian los punteros y no las cadenas reales. Por esto, la función de biblioteca `strcpy`, que fue usada en el Ejemplo 9.20, no se necesita. El programa funcionará, pues, algo más rápido que en la versión anterior.

La ejecución de este programa generará el mismo diálogo que el mostrado en el Ejemplo 9.20.

Si los elementos de un array son punteros a cadenas de caracteres, se les pueden asignar un conjunto de valores iniciales como parte de la declaración del array. En tales casos, los valores iniciales serán cadenas de caracteres, donde cada una corresponde a un elemento distinto del array. Recordar que un array debe ser declarado `static` si es inicializada dentro de una función.

Una ventaja de este esquema es que no tenemos que reservar un bloque fijo de memoria por adelantado, como se hace cuando se inicializa un array convencional. Así, si la declaración inicial incluye muchas cadenas de caracteres y algunas de ellas son relativamente cortas, se produce un ahorro sustancial en el uso de la memoria. Además, si alguna de las cadenas es especialmente larga, no hay que preocuparse de exceder la longitud máxima de cadena (el máximo número de caracteres por fila). Los arrays de este tipo se refieren a menudo como *arrays desiguales* (en inglés, *ragged arrays*).

EJEMPLO 10.27. La siguiente declaración de array aparece dentro de una función:

```
static char *nombres[10] = {
                              "PACIFICO",
                              "ATLANTICO",
                              "INDICO",
                              "CARIBE",
                              "BERING",
                              "NEGRO",
                              "ROJO",
                              "NORTE",
                              "BALTICO",
                              "CASPIO"
                           };
```

En este ejemplo nombres es un array de 10 punteros. El primer elemento del array (el primer puntero) apuntará a PACIFICO, el segundo a ATLANTICO, y así sucesivamente.

Observe que el array se declara como static para que pueda ser inicializado dentro de la función. Si la declaración del array fuera externa a todas las funciones del programa, el indicador de tipo de almacenamiento static no sería necesario.

Como la declaración del array incluye valores iniciales, no es necesario incluir una designación explícita de tamaño dentro de la declaración. El tamaño del array será automáticamente igual al número de cadenas presentes. Como resultado, la declaración anterior puede ser escrita como

```
static char *nombres[] = {
                          "PACIFICO",
                          "ATLANTICO",
                          "INDICO",
                          "CARIBE",
                          "BERING",
                          "NEGRO",
                          "ROJO",
                          "NORTE",
                          "BALTICO",
                          "CASPIO"
                        };
```

Debería quedar claro que el concepto de arrays desiguales sólo se refiere a la *inicialización* de arrays de cadenas de caracteres y no a la asignación de cadenas que se lean mediante la función scanf. Así, aplicaciones que requieran que se lean cadenas y se procesen a continuación, como en el Ejemplo 10.26, necesitan que se reserve una cantidad específica de memoria para cada elemento del array.

Las cadenas iniciales pueden ser accedidas de la forma normal mediante sus correspondientes punteros (sus correspondientes elementos del array). Estos punteros pueden reasignarse a otras constantes de cadena en otra parte del programa si es necesario.

EJEMPLO 10.28. Presentación del día del año. Desarrollaremos un programa que aceptará tres enteros, indicando el mes, día y año, y mostrará el correspondiente día de la semana, el mes, el día del mes y el año de modo más legible. Por ejemplo, supongamos que introducimos el dato 5 24 1997; esto produciría la salida Sábado, Mayo 24, 1997. A menudo se usan programas de este tipo para mostrar información almacenada en la memoria de la computadora de modo codificado.

Nuestra estrategia básica consistirá en introducir una fecha en la computadora, en la forma *mes*, *día*, *año* (mm dd aaaa), y entonces convertir este dato en el número de días relativo a algún caso base. El día de la semana de la fecha especificada se puede determinar fácilmente, siempre que conozcamos el día de la semana de la fecha base. Arbitrariamente elegimos el lunes, Enero 1, 1900 como fecha base. Convertiremos cualquier fecha posterior al 1 de enero de 1900 (realmente cualquier fecha entre el 1 de enero de 1900 y el 31 de diciembre de 2099) en su correspondiente día de la semana.

El cálculo se realiza usando las siguientes reglas empíricas:

1. Introducir valores numéricos para las variables mm, dd y aa, las cuales representan el mes, día y año, respectivamente (por ejemplo, 5 24 1997).
2. Determinar el día aproximado del año en curso, como

```
ndias = (long) (30.42 + (mm - 1)) + dd;
```

3. Si mm == 2 (febrero), incrementar el valor de ndias en 1.
4. Si mm > 2 y mm < 8 (marzo, abril, mayo, junio o julio), decrementar el valor de ndias en 1.
5. Convertir el año en el número de años transcurridos desde la fecha base; por ejemplo, aa -= 1900. Comprobar a continuación para un año bisiesto lo siguiente: si (aa % 4) == 0 y mm > 2, incrementar el valor de ndias en 1.
6. Determinar el número de ciclos completos de cuatro años detrás de la fecha base mediante aa / 4. Por cada ciclo completo de cuatro años se añade 1461 a ndias.
7. Determinar el número de años completos después del último ciclo de cuatro años mediante aa % 4. Por cada año completo sumar 365 a ndias. Entonces sumar 1, porque el primer año después del ciclo de cuatro es un año bisiesto.
8. Si ndias > 59 (si la fecha es cualquier día después del 28 de febrero de 1900), decrementar el valor de ndias en 1, ya que 1900 no es un año bisiesto. (Observe que el último año de cada siglo no es bisiesto, excepto aquellos años que son divisibles por 400. Por lo tanto 1900, el último año del siglo diecinueve, *no* es un año bisiesto, pero 2000, el último año del siglo veinte, *sí* es un año bisiesto.)
9. Determinar el día numérico de la semana correspondiente a la fecha especificada como dia = (ndias % 7).

Observe que dia == 1 corresponde o a la fecha base, que es lunes, o a otra fecha que también sea lunes. De aquí, dia == 2 será martes, dia == 3 será miércoles, ..., dia == 6 será sábado, y dia == 0 será domingo.

A continuación se muestra la función completa, llamada convertir, que realiza los pasos del 2 al 9. Observe que convertir acepta los enteros mm, dd y aa como parámetros de entrada, y devuelve la cantidad entera (ndias % 7). También observe que ndias y nciclos son variables enteras largas, mientras que el resto de las variables son enteros ordinarios.

```
int convertir(int mm, int dd, int aa)  /* convertir una fecha en el
                                          día de la semana */
{
   long ndias;   /* número de días desde el comienzo de 1900 */
   long nciclos; /* número de ciclos de 4 años después de 1900 */
   int nanios;   /* número de años después del último ciclo de 4 años */
   int dia;      /* día de la semana (0, 1, 2, 3, 4, 5 o 6) */

   /* conversiones numéricas */
   aa -= 1900;
   ndias = (long) (30.42 * (mm - 1)) + dd; /* día aproximado del
                                              año */

   if (mm == 2) ++ndias;                   /* ajuste para febrero */
   if ((mm > 2) && (mm < 8)) --ndias;      /* ajuste para marzo-
                                              julio */
   if ((aa % 4 == 0) && (mm > 2)) ++ndias; /* ajuste para el año
                                              bisiesto */

   nciclos = aa / 4;                       /* ciclos de 4 años
                                              después de 1900 */
   ndias += nciclos * 1461;                /* añadir días por ciclos
                                              de 4 años */
```

```
   nanios = aa % 4;                          /* años después del
                                                último ciclo de
                                                4 años */
   if (nanios > 0)                           /* añadir días por años
                                                después del último
                                                ciclo */
      ndias += 365 * nanios + 1;

   if (ndias > 59) --ndias;                  /* ajustar para 1900
                                                (NO año bisiesto) */

   dia = ndias % 7;

   return(dia);
}
```

Los nombres de los días de la semana se pueden almacenar como cadenas de caracteres en un array de 7 elementos; esto es,

```
static char *diasemana[] = { "Domingo", "Lunes", "Martes","Miércoles",
                             "Jueves", "Viernes", "Sábado"};
```

Cada día corresponde al valor asignado a `dia`, donde `dia = (ndias % 7)`. Los días empiezan con `Domingo` porque domingo corresponde a `dia == 0`, como se explicó anteriormente. Si la fecha base no fuera un lunes, esta ordenación particular de los días de la semana tendría que cambiarse.

Igualmente, los nombres de los meses se pueden almacenar como cadenas de caracteres en un array de 12 elementos; esto es,

```
static char *mes[] = {"Enero", "Febrero", "Marzo", "Abril", "Mayo",
                      "Junio", "Julio", "Agosto", "Septiembre",
                      "Octubre", "Noviembre", "Diciembre"};
```

Cada mes corresponde al valor de `mm - 1`.

A continuación se muestra el programa completo en C que realiza la conversión interactivamente.

```
/* convertir una fecha numérica (mm dd aaaa) en "día de la semana,
   mes, día, año"

     (por ejemplo: 5 24 1997 -> "Sábado, Mayo 24, 1997") */

#include <stdio.h>

void leerentrada(int *pm, int *pd, int *pa); /* prototipo de función */
int convertir(int mm, int dd, int aa);       /* prototipo de función */
```

```
main()

{
   int mm, dd, aa;
   int dia_semana; /* día de la semana (    0 -> Domingo,
                                            1 -> Lunes,
                                            . . .
                                            6 -> Sábado */

   static char *diasemana[] = {  "Domingo", "Lunes", "Martes",
                                 "Miércoles", "Jueves", "Viernes",
                                 "Sábado"};

   static char *mes[] = {  "Enero", "Febrero", "Marzo", "Abril", "Mayo",
                           "Junio", "Julio", "Agosto", "Septiembre",
                           "Octubre", "Noviembre", "Diciembre"};

   /* mensaje de entrada */
   printf("Rutina de conversión de fecha\n Para PARAR, introducir 0 0 0");

   leerentrada(&mm, &dd, &aa);

   /* convertir fecha a día numérico de la semana */
   while (mm > 0) {
      dia_semana = convertir(mm, dd, aa);
      printf("\n%s, %s %d, %d", diasemana[dia_semana], mes[mm-1], dd, aa);

      leerentrada(&mm, &dd, &aa);
   }
}

void leerentrada(int *pm, int *pd, int *pa)  /* leer la fecha numérica */

{
   printf("\n\nIntroducir mm dd aaaa: ");
   scanf("%d %d %d", pm, pd, pa);
   return;
}

int convertir(int mm, int dd, int aa)  /* convertir una fecha en el
                                          día de la semana */
{
   long ndias;    /* número de días desde el comienzo de 1900 */
   long nciclos;  /* número de ciclos de 4 años después de 1900 */
   int nanios;    /* número de años después del último ciclo de 4 años */
   int dia;       /* día de la semana (0, 1, 2, 3, 4, 5 o 6) */

   /* conversiones numéricas */
   aa -= 1900;
   ndias = (long) (30.42 * (mm - 1)) + dd; /* día aproximado del año */
```

```
    if (mm == 2) ++ndias;                       /* ajuste para febrero */
    if ((mm > 2) && (mm < 8)) --ndias;          /* ajuste para marzo-ju-
                                                   lio */
    if ((aa % 4 == 0) && (mm > 2)) ++ndias; /* ajuste para el año bi-
                                                   siesto */

    nciclos = aa / 4;                           /* ciclos de 4 años des-
                                                   pués de 1900 */
    ndias += nciclos * 1461;                    /* añadir días por ciclos
                                                   de 4 años */

    nanios = aa % 4;            /* años después del último ciclo de 4 años
                                   */
    if (nanios > 0)             /* añadir días por años después del último
                                   ciclo */
       ndias += 365 * nanios + 1;

    if (ndias > 59) --ndias;/* ajustar para 1900 (NO año bisiesto) */

    dia = ndias % 7;

    return(dia);
}
```

Este programa incluye un bucle que acepta repetidamente una fecha en forma de tres enteros (mm dd aaaa) y devuelve el correspondiente día y fecha de una forma más legible. El programa se ejecutará hasta que se introduzca el valor 0 para mm. Observe que el mensaje inicial del programa indica que se deben introducir tres ceros para detener la ejecución; es decir, 0 0 0. En realidad el programa sólo comprueba el valor de mm.

Una sesión interactiva típica se muestra a continuación. Como siempre, las respuestas del usuario están subrayadas.

```
Rutina de conversión de fecha
Para PARAR, introducir 0 0 0

Introducir mm dd aaaa: 10 29 1929

Martes, Octubre 29, 1929

Introducir mm dd aaaa: 8 15 1945

Miércoles, Agosto 15, 1945

Introducir mm dd aaaa: 7 20 1969

Domingo, Julio 20, 1969

Introducir mm dd aaaa: 5 24 1997

Sábado, Mayo 24, 1997
```

```
Introducir mm dd aaaa: 8 30 2010

Lunes, Agosto 30, 2010

Introducir mm dd aaaa: 4 12 2069

Viernes, Abril 12, 2069

Introducir mm dd aaaa: 0 0 0
```

10.9. PASO DE FUNCIONES A OTRAS FUNCIONES

Un puntero a una función puede ser pasado como argumento a otra función. Esto permite que una función sea transferida a otra, como si la primera función fuera una variable. Referiremos la primera función como la *función huésped* y la segunda función como la *función anfitriona*. De este modo, la huésped es pasada a la anfitriona, donde puede ser accedida. Llamadas sucesivas a la función anfitriona pueden pasar diferentes punteros (diferentes funciones huésped) a la anfitriona.

Cuando una función anfitriona acepta el nombre de una función huésped como argumento, la declaración de argumento formal debe identificar el argumento como un puntero a la función huésped. En su forma más sencilla, la declaración de argumento formal se puede escribir como

```
tipo-dato (*nombre-función)()
```

donde `tipo-dato` es el tipo de dato de la cantidad devuelta por la huésped y `nombre-función` es el nombre de la huésped. La declaración de argumento formal también se puede escribir como

```
tipo-dato (*nombre-función)(tipo 1, tipo 2, . . .)
```

o como

```
tipo-dato (*nombre-función)(tipo 1 arg 1, tipo 2 arg 2, . . .)
```

donde `tipo 1, tipo 2,`... refiere los tipos de datos de los argumentos asociados con la huésped, y `arg 1, arg 2,`... son los nombres de los argumentos asociados con la huésped.

La función huésped puede ser accedida dentro de la anfitriona mediante el operador indirección. Para hacer esto, el operador indirección debe preceder el nombre de la función huésped (el argumento formal). Tanto el operador indirección como el nombre de la función huésped deben estar encerrados entre paréntesis; esto es,

```
(*nombre-función)(argumento 1, argumento 2, . . ., argumento n);
```

donde `argumento 1, argumento 2,`..., `argumento n` son los argumentos necesarios en la llamada a la función.

Consideremos ahora la declaración de función para la función anfitriona. Ésta se puede escribir como

tipo-dato-func nombre-func(*tipo-dato-arg* (*) (*tipo 1*, *tipo 2*, ...),

|⟵ *puntero a función huésped* ⟶|

tipos de datos de los otros argumentos de la función);

donde *tipo-dato-func* es el tipo de datos de la cantidad devuelta por la función anfitriona; *nombre-func* es el nombre de la función anfitriona; *tipo-dato-arg* es el tipo de datos de la cantidad devuelta por la función huésped, y *tipo 1*, *tipo 2*, . . . son los tipos de datos de los argumentos de la función huésped. Observe que el operador indirección aparece entre paréntesis, para indicar un puntero a la función huésped. Además, le siguen los tipos de datos de los argumentos de la función huésped encerrados entre un par de paréntesis separado, para indicar que son argumentos de función.

Cuando se utiliza el prototipado completo de función, la declaración de la función anfitriona se expande de la siguiente manera:

tipo-dato-func nombre-func

(*tipo-dato-arg* (**pt-var*) (*tipo 1 arg 1*, *tipo 2 arg 2*, ...),

|⟵ *puntero a función huésped* ⟶|

tipos de datos y nombres de los otros argumentos de la función);

La notación es la misma que antes, excepto que *pt-var* es la variable puntero que apunta a la función huésped, y *tipo 1 arg 1, tipo 2 arg 2*, . . . son los tipos de datos y los nombres correspondientes de los argumentos de la función huésped.

EJEMPLO 10.29. A continuación se muestra el esquema de un programa en C. Este programa consta de cuatro funciones: main, procesar, func1 y func2. Observe que procesar es una función anfitriona para func1 y func2. Cada una de las tres funciones subordinadas devuelve un valor entero.

```
int procesar(int (*)(int, int)); /* prototipo de función (anfitriona) */
int func1(int, int);             /* prototipo de función (huésped)   */
int func2(int, int);             /* prototipo de función (huésped)   */

main()
{
   int i, j;
   . . . . .
   i = procesar(func1);  /* se pasa func1 a procesar; devuelve un va-
                            lor para i */
   . . . . .
   j = procesar(func2);  /* se pasa func2 a procesar; devuelve un va-
                            lor para j */
   . . . . .
}

procesar(int (*pf)(int, int)) /* definición de función anfitriona */
                              /* (el argumento formal es un puntero
                                 a una función)*/
```

```
{
   int a, b, c;
   . . . . .
   c = (*pf)(a, b);    /* acceso a la función pasada a esta función;
                          devuelve un valor para c */

   . . . . .
   return(c);
}

func1(int a, int b)    /* definición de función huésped */
{
   int c;

   c = . . .           /* utiliza a y b para evaluar c */

   return(c);
}

func2(int x, int y)    /* definición de función huésped */

{
   int z;

   z = . . .           /* utiliza x e y para evaluar z */

   return(z);
}
```

Observe que este programa contiene tres declaraciones de funciones. Las declaraciones para `func1` y `func2` son directas. Sin embargo, la declaración para `procesar` requiere alguna explicación. Esta declaración establece que `procesar` es una función anfitriona que devuelve un valor entero y tiene un argumento. El argumento es un puntero a una función huésped que devuelve un valor entero y tiene dos argumentos enteros. La designación del argumento para la función huésped se escribe como

```
int (*)(int, int)
```

Observe el modo en que aparece la designación de argumentos dentro de la declaración de la función anfitriona; esto es,

```
int procesar(int (*)(int, int));
```

Consideremos ahora la declaración de argumentos formales que aparece dentro de `procesar`; esto es,

```
int (*pf)(int, int);
```

Esta declaración establece que `pf` es un puntero a una función huésped. Esta función huésped devolverá un valor entero y requiere dos argumentos enteros.

Aquí tenemos otra versión del mismo esquema, utilizando el prototipado completo de función. Los cambios se muestran en negrita.

```
int procesar(int (*pf)(int a, int b));  /* declaración de función
                                           (anfitriona) */
int func1(int a, int b);                /* declaración de función
                                           (huésped) */
int func2(int a, int b);                /* declaración de función
                                           (huésped) */

main()
{
  int i, j;
  . . . . .
  i = procesar(func1);    /* se pasa func1 a procesar; devuelve un
                             valor para i */
  . . . . .
  j = procesar(func2);    /* se pasa func2 a procesar; devuelve un
                             valor para j */
  . . . . .
}

procesar(int (*pf)(int a, int b))   /* definición de función anfi-
                                       triona */
                                    /* (el argumento formal es un
                                       puntero a una función) */
{
  int a, b, c;
  . . . . .
  c = (*pf)(a, b);        /* acceso a la función pasada a esta fun-
                             ción; devuelve un valor para c */
  . . . . .
  return(c);
}

func1(int a, int b)       /* definición de función huésped */
{
  int c;

  c = . . .               /* utiliza a y b para evaluar c */

  return(c);
}

func2(int x, int y)       /* definición de función huésped */
{
  int z;
  z = . . .               /* utiliza x e y para evaluar z */

  return(z);
}
```

Los prototipos de funciones incluyen los nombres de los argumentos así como los tipos de datos de los argumentos. Además, el prototipo para `procesar` incluye ahora el nombre de la variable (`pf`) que apun-

ta a la función huésped. Observe que la declaración del argumento formal `pf` dentro de `procesar` es consistente con el prototipo de la función.

Algunos programas de aplicación se pueden formular más fácilmente en términos de paso de una función a otra. Por ejemplo, una función puede representar una ecuación matemáti-ca y la otra puede contener la estrategia computacional de resolución. En tales casos la función que representa la ecuación puede ser pasada a la función que procesa la ecuación. Esto es particularmente útil si el programa contiene diferentes ecuaciones matemáticas, de las cuales el usuario selecciona una cada vez que se ejecuta el programa.

EJEMPLO 10.30. Valor futuro de depósitos mensuales (cálculo de interés compuesto). Supongamos que una persona decide ahorrar una cantidad fija de dinero cada mes durante *n* años. Si el dinero está a un interés del *i* por ciento anual, nos preguntamos cuánto dinero tendremos dentro de *n* años (tras $12 \times n$ depósitos mensuales). La respuesta depende, por supuesto, de cuánto dinero se deposita cada mes, de la tasa de interés y de la frecuencia con que el interés se suma al depósito (frecuencia de composición). Por ejemplo, si el interés se compone anualmente, semestralmente, trimestralmente o mensualmente, la cantidad futura de dinero acumulado después de *n* años viene dada por

$$F = \frac{12A}{m}\left[\frac{(1+i/m)^{mn}-1}{i/m}\right] = 12A\left[\frac{(1+i/m)^{mn}-1}{i}\right]$$

donde *F* es la cantidad futura, *A* la cantidad de dinero depositada cada mes, *i* la tasa de interés anual (expresada como decimal) y *m* el número correspondiente de períodos por año (por ejemplo, $m = 1$ para composición anual, $m = 2$ para semestral, $m = 4$ para trimestral y $m = 12$ para mensual).

Si los períodos de composición son más cortos que los períodos de pago, como en el caso de la composición diaria, la cantidad futura se determina mediante

$$F = A\left[\frac{(1+i/m)^{mn}-1}{(1+i/m)^{m/12}-1}\right]$$

Observe que *m* tiene asignado el valor de 360 cuando el interés se compone diariamente.

Finalmente, en el caso de composición continua, la cantidad futura se determina como

$$F = A\left[\frac{e^{in}-1}{e^{i/12}-1}\right]$$

Supongamos que deseamos determinar *F* como una función de la tasa de interés anual *i*, para valores dados de *A*, *m* y *n*. Desarrollemos un programa que lea los datos necesarios dentro de `main` y entonces efectúe los cálculos en una función llamada `tabla`. Cada una de las tres fórmulas para determinar la razón *F/A* estará colocada en una de tres funciones independientes, llamadas `md1`, `md2` y `md3`, respectivamente. Por tanto, el programa constará de cinco funciones diferentes.

Cuando se llama a la función `tabla` desde `main`, uno de los argumentos pasados a `tabla` será el nombre de la función que contiene la fórmula apropiada, indicada por un parámetro de entrada (`frec`). Los valores de *A*, *m* y *n* que se leen en `main` también son pasados a `tabla` como argumentos. Se inicia un bucle dentro de `tabla`, en el cual los valores de *F* se determinan para tasas de interés desde `0.01` (1 por ciento anual) hasta `0.20` (20 por ciento anual). Los valores calculados se escriben según se van generando. A continuación se muestra el programa completo.

```
/* cálculos financieros personales */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>

/* prototipos de funciones */
void tabla(double (*pf)(double i, int m, double n), double a, int m,
          double n);
double md1(double i, int m, double n);
double md2(double i, int m, double n);
double md3(double i, int m, double n);

main()  /* calcular el valor futuro de una serie de depósitos men-
           suales */
{
  int m;        /* número de períodos de composición por año */
  double n;     /* número de años */
  double a;     /* cantidad de cada pago mensual */
  char frec;    /* indicador de frecuencia de composición */

  /* entrada de datos */
  printf("\nVALOR FUTURO DE UNA SERIE DE DEPOSITOS MENSUALES\n\n");
  printf("Cantidad de cada pago mensual: ");
  scanf("%lf", &a);
  printf("Número de años: ");
  scanf("%lf", &n);

  /* introducir frecuencia de composición */
  do {
       printf("Frecuencia de composición (A, S, T, M, D, C): ");
       scanf("%1s", &frec);
       frec = toupper(frec);    /* convertir a mayúsculas */
       if (frec == 'A') {
          m = 1;
          printf("\nComposición anual\n");
       }
       else if (frec == 'S') {
          m = 2;
          printf("\nComposición semestral\n");
       }
       else if (frec == 'T') {
          m = 4;
          printf("\nComposición trimestral\n");
       }
       else if (frec == 'M') {
          m = 12;
          printf("\nComposición mensual\n");
       }
```

```
            else if (frec == 'D') {
               m = 360;
               printf("\nComposición diaria\n");
            }
            else if (frec == 'C') {
               m = 0;
               printf("\nComposición continua\n");
            }
            else
               printf("\nERROR - Por favor repita\n\n");
      } while (frec != 'A' && frec != 'S' && frec != 'T' &&
               frec != 'M' && frec != 'D' && frec != 'C');

      /* realizar los cálculos */
      if (frec == 'C')
         tabla(md3, a, m, n);  /* composición continua */
      else if (frec == 'D')
         tabla(md2, a, m, n);  /* composición diaria */
      else
         tabla(md1, a, m, n);  /* composición anual, semestral, tri-
                                  mestral o mensual */
   }

   void tabla (double (*pf)(double i, int m, double n), double a, int m,
               double n)
   /* generador de la tabla (esta función acepta un puntero a otra
      función como argumento)

      NOTA: double (*pf)(double i, int m, double n) es un PUNTERO A
      FUNCION */

   {
      int cont;       /* contador del bucle */
      double i;       /* tasa de interés anual */
      double f;       /* valor futuro */

      printf("\nTasa de interés    Cantidad futura\n\n");
      for (cont = 1; cont <= 20; ++cont) {
          i = 0.01 * cont;
          f = a * (*pf)(i, m, n);  /* EL ACCESO A LA FUNCION SE PASA
                                      COMO UN PUNTERO */
          printf("     %2d            %.2f\n", cont, f);
      }
      return;
   }

   double md1(double i, int m, double n)
   /* depósitos mensuales, composición periódica */
   {
```

```
    double factor, razon;

    factor = 1 + i/m;
    razon = 12 * (pow(factor, m*n) - 1) / i;
    return(razon);
}

double md2(double i, int m, double n)
/* depósitos mensuales, composición diaria */

{
    double factor, razon;

    factor = 1 + i/m;
    razon = (pow(factor, m*n) - 1) / (pow(factor, m/12) - 1);
    return(razon);
}

double md3(double i, int nulo, double n)
/* depósitos mensuales, composición continua */

{
    double razon;

    razon = (exp(i*n) - 1) / (exp(i/12) - 1);
    return(razon);
}
```

Observemos los prototipos de funciones, en particular el prototipo de `tabla`. El primer argumento pasado a `tabla` es un puntero a una función huésped que recibe dos argumentos en doble precisión y un argumento entero, y devuelve una cantidad en doble precisión. Este puntero está destinado a representar a `md1`, `md2` o `md3`. Los prototipos para estas tres funciones siguen al prototipo de `tabla`. Cada función recibe dos argumentos en doble precisión y un entero y devuelve una cantidad en doble precisión como se requería.

Dentro de `main` se genera un diálogo interactivo para la entrada de datos. El programa acepta valores numéricos para `a` y `n`. También acepta una cadena de un carácter para la variable de carácter `frec`, que indica la frecuencia de composición. Los únicos caracteres permitidos que se pueden asignar a `frec` son `A`, `S`, `T`, `M`, `D` o `C` (para composición Anual, Semestral, Trimestral, Mensual, Diaria o Continua, respectivamente). Este carácter puede ser introducido como mayúscula o como minúscula, ya que es convertido a mayúscula dentro del programa. Observe que el programa hace una comprobación para que sólo se le puedan asignar los caracteres `A`, `S`, `T`, `M`, `D` o `C`.

Una vez que se determina la frecuencia de composición, se le asigna a `m` el valor adecuado. El programa accede a `tabla` pasando como argumento `md1`, `md2` o `md3`, determinado por el carácter asignado a `frec`. (Ver instrucción `if - else` múltiple al final de `main`.)

Examinemos ahora la función anfitriona `tabla`. Los últimos tres argumentos formales (`a`, `m` y `n`) están declarados como variables ordinarias de doble precisión o enteras. Sin embargo, el primer argumento formal (`pf`) está declarado como un puntero a una función huésped que acepta dos argumentos en doble precisión y un entero y devuelve una cantidad en doble precisión. Esta declaración de argumentos formales es consistente con el prototipo de función para `tabla`.

Los valores para `i` (las tasas de interés) se generan internamente dentro de `tabla`. Estos valores se determinan como `0.01 * cont`. Como `cont` va desde `1` hasta `20`, vemos que la tasa de interés varía de `0.01` a `0.20`, como se quería.

Observe la manera en que los valores de f son calculados, esto es,

```
f = a * (*pf)(i, m, n);
```

La expresión (*pf) refiere la función huésped cuyo nombre se pasa a tabla (md1, md2 o md3). Éste está acompañado por la lista requerida de argumentos, que contiene los valores actuales para i, m y n. El valor devuelto por la función huésped es multiplicado por a, y el producto se asigna a f.

Las tres funciones restantes, md1, md2 o md3, son directas. Observe que el segundo argumento en md3 es llamado nulo, ya que su valor no se usa dentro de la función. También podríamos haber hecho esto con md2, ya que el valor para m es siempre 360 días en el caso de composición diaria.

La ejecución del programa produce el siguiente diálogo interactivo:

```
VALOR FUTURO DE UNA SERIE DE DEPOSITOS MENSUALES

Cantidad de pago mensual: 100
Número de años: 3
Frecuencia de composición (A, S, T, M, D, C): p

ERROR - Por favor repita

Frecuencia de composición (A, S, T, M, D, C): m

Composición mensual

Tasa de interés   Cantidad futura

       1           3653.00
       2           3707.01
       3           3762.06
       4           3818.16
       5           3875.33
       6           3933.61
       7           3993.01
       8           4053.56
       9           4115.27
      10           4178.18
      11           4242.31
      12           4307.69
      13           4374.33
      14           4442.28
      15           4511.55
      16           4582.17
      17           4654.18
      18           4727.60
      19           4802.45
      20           4878.78
```

10.10. MÁS SOBRE DECLARACIONES DE PUNTEROS

Antes de dejar este capítulo mencionemos que las declaraciones de punteros pueden volverse complicadas y es necesario un cierto cuidado en su interpretación. Esto es especialmente cierto con declaraciones que involucren funciones o arrays.

Una dificultad es el uso dual de los paréntesis. Los paréntesis se usan para indicar funciones y para anidaciones (para establecer precedencias) dentro de declaraciones más complejas. Así la declaración

```
int *p(int a);
```

indica una función que acepta un argumento entero y devuelve un puntero a entero. Por otra parte, la declaración

```
int (*p)(int a);
```

indica un *puntero a una función* que acepta un argumento entero y devuelve un entero. En esta última declaración, el primer par de paréntesis se utiliza para anidar y el segundo par se utiliza para indicar una función.

La interpretación de declaraciones más complejas puede hacerse molesta. Por ejemplo, consideremos la declaración

```
int *(*p)(int (*a)[]);
```

En esta declaración, `(*p)( . . . )` indica un puntero a una función. Por esto, `int *(*p)( . . . )` indica un puntero a una función que devuelve un puntero a un entero. Dentro del último par de paréntesis (la especificación de los argumentos de la función), `(*a)[]` indica un puntero a un array. Como resultado, `int (*a)[]` representa un puntero a un array de enteros. Uniendo las piezas, `(*p)(int (*a)[])` representa un puntero a un array cuyo argumento es un puntero a un array de enteros. Y finalmente, la declaración completa

```
int *(*p)(int (*a)[]);
```

representa un puntero a una función que acepta un puntero a un array de enteros como argumento y devuelve un puntero a entero.

Recordar que un paréntesis izquierdo siguiendo inmediatamente a un nombre de identificador indica que el identificador representa una función. Análogamente, un corchete izquierdo siguiendo inmediatamente a un nombre de identificador indica que el identificador representa un array. Los paréntesis que identifican las funciones y los corchetes que identifican arrays tienen mayor precedencia que el operador unario indirección (ver Apéndice C). Por tanto, serán necesarios paréntesis adicionales cuando se declare un puntero a una función o un puntero a un array.

El siguiente ejemplo proporciona ilustraciones de esto.

EJEMPLO 10.31. A continuación se muestran diversas declaraciones que involucran a punteros. Las declaraciones están ordenadas de simples a complejas.

```
int *p;                     /* p es un puntero a una cantidad entera */
int *p[10];                 /* p es un array de 10 punteros a ente-
                               ros */
int (*p)[10];               /* p es un puntero a un array de 10 ente-
                               ros */
int *p(void);               /* p es una función que devuelve un puntero
                               a entero */
int p(char *a);             /* p es una función que acepta un argumento
                               que es un puntero a carácter y devuelve
                               un entero */
int *p(char *a);            /* p es una función que acepta un argumento
                               que es un puntero a carácter y devuelve
                               un puntero a un entero */
int (*p)(char *a);          /* p es un puntero a una función que acepta
                               un argumento que es un puntero a carác-
                               ter y devuelve un entero */
int (*p(char *a))[10];      /* p es una función que acepta un argumen-
                               to que es un puntero a carácter y de-
                               vuelve un puntero a un array de diez
                               enteros */
int p(char (*a)[]);         /* p es una función que acepta un argumento
                               que es un puntero a un array de caracte-
                               res y devuelve un entero */
int p(char *a[]);           /* p es una función que acepta un argumento
                               que es un array de punteros a caracteres
                               y devuelve un entero */
int *p(char a[]);           /* p es una función que acepta un argumento
                               que es un array de caracteres y devuelve
                               un puntero a entero */
int *p(char (*a)[]);        /* p es una función que acepta un argumento
                               que es un puntero a un array de caracte-
                               res y devuelve un puntero a entero */
int *p(char *a[]);          /* p es una función que acepta un argu-
                               mento que es un array de punteros a
                               caracteres y devuelve un puntero a ente-
                               ro */
int (*p)(char (*a)[]);      /* p es un puntero a una función que acepta
                               un argumento que es un puntero a un array
                               de caracteres y devuelve un entero */
int *(*p)(char (*a)[]);     /* p es un puntero a una función que acepta
                               un argumento que es un puntero a un array
                               de caracteres y devuelve un puntero a
                               entero */
int *(*p)(char *a[]);       /* p es un puntero a una función que acepta
                               un argumento que es un array de punteros
                               a caracteres y devuelve un puntero a en-
                               tero */
int (*p[10])(void);         /* p es un array de 10 punteros a función;
                               cada función devuelve un entero */
```

```
int (*p[10])(char a);     /* p es un array de 10 punteros a función;
                             cada función acepta un argumento que es un
                             carácter y devuelve un entero */
int *(*p[10])(char a);    /* p es un array de 10 punteros a función;
                             cada función acepta un argumento que es
                             un carácter y devuelve un puntero a en-
                             tero */
int *(*p[10])(char *a);   /* p es un array de 10 punteros a función;
                             cada función acepta un argumento que es
                             un puntero a carácter y devuelve un pun-
                             tero a entero */
```

CUESTIONES DE REPASO

10.1. En la versión de C disponible en su computadora particular, ¿cuántas celdas de memoria son necesarias para almacenar un carácter? ¿Un entero? ¿Un entero largo? ¿Una cantidad en coma flotante? ¿Una cantidad en doble precisión?

10.2. ¿Qué se entiende por la dirección de una celda de memoria? ¿Cómo están normalmente numeradas las direcciones?

10.3. ¿Cómo se determina la dirección de una variable?

10.4. ¿Qué tipo de información representa una variable puntero?

10.5. ¿Cuál es la relación entre la dirección de una variable `v` y la variable puntero correspondiente `pv`?

10.6. ¿Cuál es la finalidad del operador indirección? ¿A qué tipo de operandos se debe aplicar?

10.7. ¿Cuál es la relación entre el dato representado por la variable `v` y la variable puntero correspondiente `pv`?

10.8. ¿Qué precedencia se le asigna a los operadores unarios comparados con los operadores multiplicación, división y resto? ¿En qué orden se evalúan los operadores unarios?

10.9. ¿Puede actuar el operador dirección sobre una expresión aritmética como `2 * (u + v)`? Explicar las razones de la respuesta.

10.10. ¿Puede una expresión involucrar que un operador indirección aparezca a la izquierda de una instrucción de asignación? Explicarlo.

10.11. ¿Qué tipos de objetos pueden asociarse con variables puntero?

10.12. ¿Cómo se declara una variable puntero? ¿Cuál es la finalidad del tipo de datos incluido en la declaración?

10.13. ¿De qué manera puede incluirse la asignación de un valor inicial en la declaración de una variable puntero?

10.14. ¿Se asignan alguna vez valores enteros a una variable puntero? Explicarlo

10.15. ¿Por qué es a veces deseable pasar un puntero como argumento de una función?

10.16. Supongamos que una función recibe un puntero como argumento. Explicar cómo se escribe el prototipo de la función. En particular, explicar cómo se representa el tipo de datos del argumento puntero.

10.17. Supongamos que una función recibe un puntero como argumento. Explicar cómo se declara el argumento puntero en la definición de la función.

10.18. ¿Cuál es la relación entre el nombre de un array y un puntero? ¿Cómo es interpretado el nombre de un array cuando aparece como argumento de una función?

10.19. Supongamos que un argumento formal dentro de la definición de una función es un array. ¿Cómo puede declararse el array dentro de la función?

10.20. ¿Cómo puede pasarse una parte de un array a una función?

10.21. ¿Cómo puede una función devolver un puntero a la rutina que la llama?

10.22. Describir dos formas diferentes de especificar la dirección del elemento de un array.

10.23. ¿Por qué el valor del índice de un array se refiere a veces como un desplazamiento cuando el índice es parte de una expresión que indica la dirección de un elemento del array?

10.24. Describir dos modos diferentes de acceder a un elemento de un array. Comparar la respuesta con la de la cuestión 10.22.

10.25. ¿Puede asignársele una dirección a un nombre de array o a un elemento del array? ¿Puede asignársele una dirección a una variable puntero cuyo objeto es un array?

10.26. Supongamos que se define un array numérico en términos de una variable puntero. ¿Pueden inicializarse los elementos del array?

10.27. Supongamos que se define un array de caracteres en términos de una variable puntero. ¿Pueden inicializarse los elementos del array? Comparar la respuesta con la de la cuestión previa.

10.28. ¿Qué se entiende por asignación dinámica de memoria? ¿Qué función de biblioteca se utiliza para asignar memoria dinámicamente? ¿Cómo se especifica el tamaño del bloque de memoria? ¿Qué clase de información es devuelta por la función de biblioteca?

10.29. Supongamos que una cantidad entera es sumada o restada de una variable puntero. ¿Cómo será interpretada la suma o la diferencia?

10.30. ¿Bajo qué condiciones puede una variable puntero ser restada de otra? ¿Cómo se interpretará esta diferencia?

10.31. ¿Bajo qué condiciones se pueden comparar dos variables puntero? ¿Bajo qué condiciones son útiles tales comparaciones?

10.32. ¿Cómo se define un array multidimensional en términos de un puntero a una colección de arrays contiguos de más baja dimensionalidad?

10.33. ¿Cómo se puede utilizar el operador indirección para acceder a un elemento del array multidimensional?

10.34. ¿Cómo se define un array multidimensional en términos de un array de punteros? ¿Qué representa cada puntero? ¿En qué difiere esta definición de un puntero a una colección de arrays contiguos de más baja dimensionalidad?

10.35. ¿Cómo se puede utilizar un array unidimensional de punteros para representar una colección de cadenas de caracteres?

10.36. Si se almacenan varias cadenas de caracteres en un array unidimensional de punteros, ¿cómo se puede acceder a una cadena individual?

10.37. Si se almacenan varias cadenas de caracteres en un array unidimensional de punteros, ¿qué sucede si las cadenas son reordenadas? ¿Son movidas realmente las cadenas de caracteres a diferentes posiciones dentro del array?

10.38. ¿Bajo qué condiciones se pueden inicializar los elemento de un array multidimensional si el array se define en términos de un array de punteros?

10.39. Cuando se transfiere una función a otra, ¿cuál es el significado de la función huésped? ¿Cuál el de la función anfitriona?

10.40. Supongamos que un argumento formal dentro de una definición de función anfitriona es un puntero a otra función. ¿Cómo se declara el argumento formal? Dentro de la declaración, ¿a qué tipo de datos refiere?

10.41. Supongamos que un argumento formal `p` dentro de una definición de función anfitriona es un puntero a la función huésped `q`. ¿Cómo se declara el argumento formal dentro de `p`? En esta declaración, ¿a qué tipo de datos refiere? ¿Cómo se accede a la función `q` desde la función `p`?

10.42. Supongamos que `p` es una función anfitriona y uno de los argumentos de `p` es un puntero a la función `q`. ¿Cómo se debería escribir la declaración para `p` si se utiliza el prototipo de función completo?

10.43. ¿Para qué tipo de aplicaciones es particularmente útil el paso de una función a otra.

PROBLEMAS

10.44. Explicar el sentido de cada una de las siguientes declaraciones:

a) `int *px;`

b) `float a, b;`
`float *pa, *pb;`

c) `float a = -0.167;`
`float *pa = &a;`

d) `char c1, c2, c3;`
`char *pc1, *pc2, *pc3 = &c1;`

e) `double func(double *a, double *b, int *c);`

f) `double *func(double *a, double *b, int *c);`

g) `double (*a)[12];`

h) `double *a[12];`

i) `char *a[12];`

j) `char *d[4] = {"norte", "sur", "este", "oeste"};`

k) `long (*p)[10][20];`

l) `long *p[10][20];`

m) `char muestra(int (*pf)(char a, char b));`

n) `int (*pf)(void);`

o) `int (*pf)(char a, char b);`

p) `int (*pf)(char *a, char *b);`

10.45. Escribir una declaración apropiada para cada una de las siguientes situaciones:

a) Declarar dos punteros cuyos objetos sean las variables enteras `i` y `j`.

b) Declarar un puntero a una cantidad en coma flotante y otro a una en doble precisión.

c) Declarar una función que acepte dos argumentos enteros y devuelva un puntero a entero largo.

d) Declarar una función que acepte dos argumentos y devuelva un entero largo. Cada argumento será un puntero a entero.

e) Declarar un array unidimensional de elementos en coma flotante usando la notación de punteros.

f) Declarar un array bidimensional de elementos en coma flotante, con `15` filas y `30` columnas, usando notación de punteros.

g) Declarar un array de cadenas de caracteres cuyos valores iniciales sean "rojo", "verde" y "azul".

h) Declarar una función que acepte otra función como argumento y devuelva un puntero a carácter. La función pasada como argumento aceptará como argumento un entero y devolverá otro entero.

i) Declarar un puntero a una función que acepte tres enteros como argumentos y devuelva una cantidad en coma flotante.

j) Declarar un puntero a una función que acepte tres punteros a enteros como argumentos y devuelva un puntero a una cantidad en coma flotante.

10.46. Un programa en C contiene las siguientes instrucciones:

```
char u, v = 'A';
char *pu, *pv = &v;

. . . . .

*pv = v + 1;
u = *pv + 1;
pu = &u;
```

Supongamos que cada carácter ocupa un byte de memoria. Si el valor asignado a `u` se almacena en la dirección `F8C` (hexadecimal) y el valor asignado a `v` se almacena en `F8D`, entonces:

a) ¿Qué valor es representado por `&v`?
b) ¿Qué valor es asignado a `pv`?
c) ¿Qué valor es representado por `*pv`?
d) ¿Qué valor es asignado a `u`?
e) ¿Qué valor es representado por `&u`?
f) ¿Qué valor es asignado a `pu`?
g) ¿Qué valor es representado por `*pu`?

10.47. Un programa en C contiene las siguientes instrucciones:

```
int i, j = 25;
int *pi, *pj = &j;

. . . . .

*pj = j + 5;
i = *pj + 5;
pi = pj;
*pi = i + j;
```

Supongamos que cada cantidad entera ocupa 2 bytes de memoria. Si el valor asignado a `i` empieza en la dirección `F9C` (hexadecimal) y el valor asignado a `j` empieza en `F9E`, entonces:

a) ¿Qué valor es representado por `&i`?
b) ¿Qué valor es representado por `&j`?
c) ¿Qué valor es asignado a `pj`?
d) ¿Qué valor es asignado a `*pj`?
e) ¿Qué valor es asignado a `i`?
f) ¿Qué valor es representado por `pi`?
g) ¿Qué valor final es asignado a `*pi`?
h) ¿Qué valor es representado por `(pi + 2)`?
i) ¿Qué valor es representado por la expresión `(*pi + 2)`?
j) ¿Qué valor es representado por la expresión `*(pi + 2)`?

10.48. Un programa en C contiene las siguientes instrucciones:

```
float a = 0.001, b = 0.003;
float c, *pa, *pb;

pa = &a;
*pa = 2 * a;
pb = &b;
c = 3 * (*pb - *pa);
```

Supongamos que cada cantidad en coma flotante ocupa 4 bytes de memoria. Si el valor asignado a `a` empieza en la dirección `1130` (hexadecimal), el valor asignado a `b` empieza en `1134` y el valor asignado a `c` empieza en `1138`, entonces:

a) ¿Qué valor es asignado a `&a`?
b) ¿Qué valor es asignado a `&b`?
c) ¿Qué valor es asignado a `&c`?
d) ¿Qué valor es asignado a `pa`?
e) ¿Qué valor es representado por `*pa`?
f) ¿Qué valor es representado por `&(*pa)`?
g) ¿Qué valor es asignado a `pb`?
h) ¿Qué valor es representado por `*pb`?
i) ¿Qué valor es asignado a `c`?

10.49. A continuación se muestra el esquema de la estructura de un programa en C.

```
int func1(char a, char b);
int func2(char *pa, char *pb);

main()
{
  char a = 'X';
  char b = 'Y';
  int i, j;

  . . . . .
```

```
    i = func1(a, b);
    printf("a=%d   b=%d\n", a, b);
    . . . . .
    j = func2(&a, &b);
    printf("a=%d   b=%d\n", a, b);
}

int func1(char c1, char c2)
{
    c1 = 'P';
    c2 = 'Q';
    . . . . .
    return((c1 < c2) ? c1 : c2);
}

int func2(char *c1, char *c2)
{
    *c1 = 'P';
    *c2 = 'Q';
    . . . . .
    return((*c1 == *c2) ? *c1 : *c2);
}
```

a) Dentro de `main`, ¿qué valor es asignado a `i`?
b) ¿Qué valor es asignado a `j`?
c) ¿Qué valores son escritos por la primera instrucción `printf`?
d) ¿Qué valores son escritos por la segunda instrucción `printf`?

Suponer caracteres ASCII.

10.50. El esquema de la estructura de un programa en C se muestra a continuación.

```
void func(int *p);

main()
{
    static int a[5] = {10, 20, 30, 40, 50};
    . . . . .
    func(a);
    . . . . .
}

void func(int *p)
{
    int i, suma = 0;
    for (i = 0; i < 5; ++i)
        suma += *(p + i);
    printf("suma=%d", suma);
    return;
}
```

a) ¿Qué tipo de argumento se pasa a `func`?
b) ¿Qué tipo de información devuelve `func`?
c) ¿Qué tipo de argumento formal se devuelve dentro de `func`?
d) ¿Cuál es la finalidad del bucle `for` que aparece dentro de `func`?
e) ¿Qué valor es escrito por la instrucción `printf` dentro de `func`?

10.51. A continuación se muestra el esquema de la estructura de un programa en C.

```
void func(int *p);

main()
{
  static int a[5] = {10, 20, 30, 40, 50};
  . . . . .
  func(a + 3);
  . . . . .
}

void func(int  *p)
{
  int i, suma = 0;
  for (i = 0; i < 2; ++i)
      suma += *(p + i);
  printf("suma=%d", suma);
  return;
}
```

a) ¿Qué tipo de argumento se pasa a `func`?
b) ¿Qué tipo de información devuelve `func`?
c) ¿Qué información se pasa realmente a `func`?
d) ¿Cuál es la finalidad del bucle `for` que aparece dentro de `func`?
e) ¿Qué valor es escrito por la instrucción `printf` dentro de `func`?

Comparar las respuestas con las del problema anterior. ¿En qué se diferencian estos dos esquemas de programa?

10.52. A continuación se muestra el esquema de la estructura de un programa en C.

```
int *func(int *p);

main()
{
  static int a[5] = {10, 20, 30, 40, 50};
  int *ptmax;
  . . . . .

  ptmax = func(a);
  printf("max=%d", *ptmax);
  . . . . .
}
```

```
int *func(int *p)
{
   int i, imax, max = 0;
   for (i = 0; i < 5; ++i)
        if (*(p + i) > max) {
           max = *(p + i);
           imax = i;
        }
   return(p + imax);
}
```

a) Dentro de `main`, ¿qué es `ptmax`?
b) ¿Qué tipo de información devuelve `func`?
c) ¿Qué se asigna a `ptmax` cuando se accede a la función?
d) ¿Cuál es la finalidad del bucle `for` que aparece dentro de `func`?
e) ¿Qué valor es escrito por la instrucción `printf` dentro de `main`?

Comparar las respuestas con las de los dos problemas anteriores. ¿En qué se diferencian estos esquemas?

10.53. Un programa en C contiene la siguiente declaración:

```
static int x[8] = {10, 20, 30, 40, 50, 60, 70, 80};
```

a) ¿Cuál es el significado de `x`?
b) ¿Cuál es el significado de `(x + 2)`?
c) ¿Cuál es el valor de `*x`?
d) ¿Cuál es el valor de `(*x + 2)`?
e) ¿Cuál es el valor de `*(x + 2)`?

10.54. Un programa en C contiene la siguiente declaración:

```
static float tabla[2][3] = {
                              {1.1, 1.2, 1.3},
                              {2.1, 2.2, 2.3}
                           };
```

a) ¿Cuál es el significado de `tabla`?
b) ¿Cuál es el significado de `(tabla + 1)`?
c) ¿Cuál es el significado de `*(tabla + 1)`?
d) ¿Cuál es el significado de `(*(tabla + 1) + 1)`?
e) ¿Cuál es el significado de `(*(tabla) + 1)`?
f) ¿Cuál es el valor de `*(*(tabla + 1) + 1)`?
g) ¿Cuál es el valor de `*(*(tabla) + 1)`?
h) ¿Cuál es el valor de `*(*(tabla + 1))`?
i) ¿Cuál es el valor de `*(*(tabla) + 1) + 1`?

10.55. Un programa en C contiene la siguiente declaración:

```
static char *color[6] = {"rojo", "verde", "azul", "blanco",
                         "negro", "amarillo"};
```

a) ¿Cuál es el significado de `color`?
b) ¿Cuál es el significado de `(color + 2)`?
c) ¿Cuál es el valor de `*color`?
d) ¿Cuál es el valor de `*(color + 2)`?
e) ¿En qué se diferencian `color[5]` y `*(color + 5)`?

10.56. A continuación se muestra el esquema de la estructura de un programa en C.

```
float uno(float x, float y);
float dos(float x, float y);
float tres(float (*pt)(float x, float y));

main()
{
   float a, b;

   . . . . .

   a = tres(uno);

   . . . . .

   b = tres(dos);

   . . . . .

}

float uno(float x, float y)
{
   float z;

   z = . . . . .

   return(z);
}

float dos(float x, float y)
{
   float r;

   r = . . . . .

   return(r);
}

float tres(float (*pt)(float x, float y))
{
   float a, b, c;

   . . . . .

   c = (*pt)(a, b);

   . . . . .

   return(c);
}
```

a) Interpretar cada uno de los prototipos de funciones.
b) Interpretar las definiciones de las funciones `uno` y `dos`.
c) Interpretar la definición de la función `tres`. ¿En qué se diferencia `tres` de `uno` y `dos`?
d) ¿Qué ocurre dentro de `main` cada vez que se accede a `tres`?

10.57. A continuación se muestra el esquema de la estructura de un programa en C.

```
float uno(float *px, float *py);
float dos(float *px, float *py);
float *tres(float (*pt)(float *px, float *py));

main()
{
  float *pa, *pb;

  . . . . .

  pa = tres(uno);

  . . . . .

  pb = tres(dos);
  . . . . .
}

float uno(float *px, float *py)
{
  float z;

  z = . . . . .

  return(z);
}

float dos(float *pp, float *pq)
{
  float r;

  r = . . . . .

  return(r);
}

float *tres(float (*pt)(float *px, float *py))
{
  float a, b, c;

  . . . . .

  c = (*pt)(&a, &b);

  . . . . .

  return(&c);
}
```

a) Interpretar cada uno de los prototipos de funciones.
b) Interpretar las definiciones de las funciones `uno` y `dos`.
c) Interpretar la definición de la función `tres`. ¿En qué se diferencia `tres` de `uno` y `dos`?
d) ¿Qué ocurre dentro de `main` cada vez que se accede a `tres`?
e) ¿En qué se diferencia este esquema de programa del mostrado en el último ejemplo?

10.58. Explicar el propósito de cada una de las siguientes declaraciones:

a) `float (*x)(int *a);`
b) `float (*x(int *a))[20];`
c) `float x(int (*a)[]);`
d) `float x(int *a[]);`
e) `float *x(int a[]);`
f) `float *x(int (*a)[]);`
g) `float *x(int *a[]);`
h) `float (*x)(int (*a)[]);`
i) `float *(*x)(int *a[]);`
j) `float (*x[20])(int a);`
k) `float *(*x[20])(int *a);`

10.59. Escribir una declaración apropiada para cada una de las siguientes situaciones con punteros:

a) Declarar una función que acepte un argumento que es un puntero a un entero y devuelva un puntero a un array de seis caracteres.
b) Declarar una función que acepte un argumento que es un puntero a un array de enteros y devuelva un carácter.
c) Declarar una función que acepte un argumento que es un array de punteros a enteros y devuelva un carácter.
d) Declarar una función que acepte un argumento que es un array de enteros y devuelva un puntero a carácter.
e) Declarar una función que acepte un argumento que es un puntero a un array de enteros y devuelva un puntero a carácter.
f) Declarar una función que acepte un argumento que es un array de punteros a enteros y devuelva un puntero a carácter.
g) Declarar un puntero a una función que acepte un argumento que es un puntero a un array de enteros y devuelva un carácter.
h) Declarar un puntero a una función que acepte un argumento que es un puntero a un array de enteros y devuelva un puntero a carácter.
i) Declarar un puntero a una función que acepte un argumento que es un array de punteros a enteros y devuelva un puntero a carácter.
j) Declarar un array de 12 punteros a funciones. Cada función aceptará como argumentos dos cantidades en doble precisión y devolverá un número en doble precisión.
k) Declarar un array de 12 punteros a funciones. Cada función aceptará como argumentos dos cantidades en doble precisión y devolverá un puntero a un número en doble precisión.
l) Declarar un array de 12 punteros a funciones. Cada función aceptará como argumentos dos punteros a cantidades en doble precisión y devolverá un puntero a un número en doble precisión.

PROBLEMAS DE PROGRAMACIÓN

10.60. Modificar el programa mostrado en el Ejemplo 10.1 como sigue:

a) Usar datos en coma flotante en vez de enteros. Asignar el valor inicial 0.3 a `u`.
b) Usar datos en doble precisión en vez de enteros. Asignar el valor inicial 0.3×10^{45} a `u`.
c) Usar caracteres en vez de enteros. Asignar un valor inicial de `'C'` a `u`.

Ejecutar cada modificación y comparar los resultados con los del Ejemplo 10.1. Asegurarse de modificar las instrucciones `printf` de modo adecuado.

10.61. Modificar el programa mostrado en el Ejemplo 10.3 como sigue:

a) Usar datos en coma flotante en vez de enteros. Asignar el valor inicial 0.3 a `v`.
b) Usar datos en doble precisión en vez de enteros. Asignar el valor inicial 0.3×10^{45} a `v`.
c) Usar caracteres en vez de enteros. Asignar un valor inicial de `'C'` a `v`.

Ejecutar cada modificación y comparar los resultados con los del Ejemplo 10.3. Asegurarse de modificar las instrucciones `printf` de modo adecuado.

10.62. Modificar el programa mostrado en el Ejemplo 10.7 de modo que se pase un array unidimensional de caracteres a `func1`. Borrar `func2` y todas sus referencias. Inicialmente asignar la cadena "rojo" al array dentro de `main`. Reasignar la cadena "verde" al array dentro de `func1`. Ejecutar el programa y comparar los resultados con los mostrados en el Ejemplo 10.7. Acordarse de modificar las instrucciones `printf` de forma adecuada.

10.63. Modificar el programa mostrado en el Ejemplo 10.8 (análisis de una línea de texto) de modo que se cuenten también el número de palabras y el número total de caracteres en la línea de texto. (*Nota:* una nueva palabra puede ser reconocida por la presencia de un espacio en blanco seguido de otro carácter no blanco.) Comprobar el programa usando la línea de texto dada en el Ejemplo 10.8.

10.64. Modificar el programa mostrado en el Ejemplo 10.8 (análisis de una línea de texto) de modo que se puedan procesar varias líneas de texto. Primero introducir y almacenar todas las líneas. Luego determinar el número de vocales, consonantes, dígitos, espacios en blanco y «otros» caracteres para cada línea. Finalmente, determinar la media del número de vocales por línea, consonantes por línea, etc. Escribir y ejecutar el programa de dos formas diferentes:

a) Almacenar las líneas de texto en un array bidimensional de caracteres.
b) Almacenar las líneas de texto como cadenas de caracteres de longitud máxima no especificada. Mantener un puntero a cada cadena dentro de un array unidimensional de punteros.

En cada caso, identificar la última línea de texto de un modo predeterminado (por ejemplo, introduciendo la cadena `"FIN"`). Comprobar el programa usando varias líneas de texto de su elección.

10.65. Modificar el programa mostrado en el Ejemplo 10.12 de modo que los elementos de `x` sean enteros largos en vez de enteros ordinarios. Ejecutar el programa y comparar los resultados con los del Ejemplo 10.12. Recuérdese modificar la instrucción `printf` para acomodarla a los enteros largos.

10.66. Modificar el programa mostrado en el Ejemplo 10.16 de modo que se puedan efectuar cualquiera de las reordenaciones siguientes:

a) Menor a mayor, en valor absoluto
b) Menor a mayor, algebraico
c) Mayor a menor, en valor absoluto
d) Mayor a menor, algebraico

Utilizar notación de punteros para representar enteros individuales, como en el Ejemplo 10.16. (Recordar que en el Ejemplo 9.13 se presentó una versión con arrays de este problema.) Incluir un menú que permitirá al usuario seleccionar qué reordenación será usada cada vez que se ejecute el programa. Comprobar el programa utilizando los 10 valores siguientes:

4.7	–8.0
–2.3	11.4
12.9	5.1
8.8	–0.2
6.0	–14.7

10.67. Modificar el programa mostrado en el ejemplo 10.22 (suma de dos tablas de números) de modo que cada elemento en la tabla `c` sea el mayor de los correspondientes elementos en las tablas `a` y `b` (en vez de la suma de los elementos correspondientes de `a` y `b`). Representar cada tabla (cada array) como un puntero a un grupo de arrays unidimensionales, como en el Ejemplo 10.22. Utilizar notación de punteros para acceder a los elementos individuales de la tabla. Comprobar el programa usando las tablas del Ejemplo 9.19. (Se debería experimentar con este programa usando diferentes formas de representar los arrays y los elementos individuales de los arrays.)

10.68. Repetir el problema anterior representando cada tabla (cada array) como un array unidimensional de punteros, como se discute en el Ejemplo 10.24.

10.69. Modificar el programa mostrado en el Ejemplo 10.26 (reordenación de una lista de cadenas de caracteres) de modo que la lista de cadenas pueda ser reordenada en orden alfabético o en orden alfabético inverso. Utilizar notación de punteros para representar el comienzo de cada cadena. Incluir un menú que permita al usuario seleccionar qué reordenación se realiza cada vez que se ejecuta el programa. Comprobar el programa con los datos del Ejemplo 9.20.

10.70. Modificar el programa mostrado en el Ejemplo 10.28 (presentación del día del año) de modo que pueda determinar el número de días entre dos fechas, suponiendo que ambas fechas son posteriores a la fecha base del 1 de enero de 1900. (*Sugerencia:* Determinar el número de días entre la primera fecha y la fecha base; hacer a continuación lo mismo para la segunda fecha y finalmente calcular la diferencia entre los valores calculados.)

10.71. Modificar el programa mostrado en el Ejemplo 10.30 (cálculo de interés compuesto) de modo que genere una tabla de valores futuros (valores de `F`) para varias tasas de interés, utilizando diferentes frecuencias de composición. Suponer que `A` y `n` son valores a introducir. Mostrar la salida de la siguiente manera:

```
                         A = . . .
                         n = . . .

Tasa de interés =        5%  6%  7%  8%  9% 10% 11% 12% 13% 14% 15%

Frecuencia de
composición

Anual                    __  __  __  __  __  __  __  __  __  __  __
Semestral                __  __  __  __  __  __  __  __  __  __  __
Trimestral               __  __  __  __  __  __  __  __  __  __  __
Mensual                  __  __  __  __  __  __  __  __  __  __  __
Diaria                   __  __  __  __  __  __  __  __  __  __  __
Continua                 __  __  __  __  __  __  __  __  __  __  __
```

Observe que las primeras cuatro filas se generan con una misma función con diferentes argumentos y cada una de las dos últimas filas se genera con una función diferente.

10.72. Modificar el programa mostrado en el Ejemplo 10.30 (cálculo de interés compuesto) de modo que genere una tabla de valores futuros (valores `F`) para varios períodos de tiempo y diferentes frecuencias de composición. Suponer que `A` e `i` son valores a introducir. Mostrar la salida de la siguiente manera.

```
                         A = . . .
                         i = . . .

Período de tiempo (n)=      1   2   3   4   5   6   7   8   9   10

Frecuencia de
composición

Anual                       __  __  __  __  __  __  __  __  __  __
Semestral                   __  __  __  __  __  __  __  __  __  __
Trimestral                  __  __  __  __  __  __  __  __  __  __
Mensual                     __  __  __  __  __  __  __  __  __  __
Diaria                      __  __  __  __  __  __  __  __  __  __
Continua                    __  __  __  __  __  __  __  __  __  __
```

Observe que las cuatro primeras filas se generan con una misma función con diferentes argumentos y cada una de las dos últimas filas son generadas por funciones diferentes.

10.73. Repetir el problema anterior, pero transponiendo la tabla de modo que cada fila represente un valor diferente para `n` y cada columna represente una frecuencia de composición distinta. Considerar valores enteros de `n` en el rango de `1` a `50`. Observe que la tabla tendrá `50` filas y `6` columnas. (*Sugerencia:* Generar la tabla por columnas, almacenando cada columna en un array bidimensional. Mostrar el array completo cuando se hayan generado todos los valores.)
Comparar el esfuerzo de programación de este problema con el requerido para el problema anterior.

10.74. Los Ejemplos 9.8 y 9.9 presentan programas para calcular la media de una lista de números y luego calcular las desviaciones respecto de la media. Ambos programas hacen uso de arrays unidimensionales en coma flotante. Modificar ambos programas utilizando notación de punteros.

(Observe que el programa del Ejemplo 9.9 incluye la asignación de valores iniciales a los elementos del array.) Comprobar ambos programas usando los datos dados en los ejemplos.

10.75. Modificar el programa dado en el Ejemplo 9.14 (generador de «pig latin») de modo que utilice arrays de caracteres. Modificar el programa de modo que utilice notación de punteros. Comprobar el programa usando varias líneas de texto de su elección.

10.76. Escribir un programa completo en C, utilizando la notación de punteros en vez de arrays, para los siguientes programas tomados del final del Capítulo 9.

a) Problema 9.39 (leer una línea de texto, almacenarla en la memoria de la computadora y escribirla hacia atrás).
b) Problema 9.40 (procesar un conjunto de calificaciones de exámenes de alumnos). Comprobar el programa usando los datos dados en el Problema 9.40.
c) Problema 9.42 (procesar un conjunto de calificaciones ponderadas de exámenes y calcular la desviación de la media de cada alumno respecto a la media general de la clase). Comprobar el programa con los datos del Problema 9.40.
d) Problema 9.44 (generar una tabla de factores de interés compuesto).
e) Problema 9.45 (cambiar de una moneda extranjera a otra).
f) Problema 9.46 (determinar la capital de un país especificado o el país cuya capital es especificada). Comprobar el programa utilizando la lista de países y capitales dada en el Problema 9.46.
g) Problema 9.47(*a*) (multiplicación de matriz/vector). Comprobar el programa con los datos dados en el problema 9.47(*a*).
h) Problema 9.47(*b*) (multiplicación de matrices). Comprobar el programa con los datos dados en el problema 9.47(*b*).
i) Problema 9.47(*d*) (interpolación de Lagrange). Comprobar el programa con los datos dados en el problema 9.47(*d*).
j) Problema 9.48(*a*) («blackjack»).
k) Problema 9.48(*b*) (ruleta).
l) Problema 9.48(*c*) (BINGO).
m) Problema 9.49 (codificar y decodificar una línea de texto).

10.77. Escribir un programa completo en C, utilizando notación de punteros, que genere las siguientes tres columnas:

$t \qquad ae^{bt}$ sen $ct \qquad ae^{bt}$ cos ct

Estructurar el programa de la siguiente manera: escribir dos funciones especiales, `f1` y `f2`, donde `f1` evalúa la cantidad ae^{bt} sen ct y `f2` evalúa ae^{bt} cos ct. Leer los valores de a, b y c en `main`, y luego llamar a la función `gen_tabla`, que generará la tabla real. Pasar `f1` y `f2` a `gen_tabla` como argumentos.
Comprobar el programa utilizando los valores $a = 2$, $b = -0.1$, $c = 0.5$, donde los valores de t son 1, 2, 3, ..., 60.

CAPÍTULO 11

Estructuras y uniones

En el Capítulo 9 hemos estudiado el array, que es una estructura de datos cuyos elementos son todos del mismo tipo. Ahora fijamos nuestra atención sobre la *estructura*, que es una estructura de datos cuyos elementos individuales pueden ser de distinto tipo. Así, una estructura puede contener elementos enteros, en coma flotante y caracteres. Punteros, arrays y otras estructuras pueden ser también incluidas como elementos dentro de una estructura. A los elementos individuales de una estructura se les denomina *miembros*.

Este capítulo se ocupa del uso de estructuras en un programa en C. Veremos cómo se definen las estructuras y cómo se accede y se procesan sus miembros dentro de un programa. También se examina la relación entre estructuras y punteros, arrays y funciones.

Estrechamente relacionada con la estructura está la *unión*, que también contiene múltiples miembros. Sin embargo, a diferencia de la estructura, los miembros de una unión comparten el mismo área de almacenamiento, incluso cuando los miembros son de diferente tipo. Así, la unión permite que varios datos diferentes se almacenen en la misma parte de memoria de la computadora en distintos tiempos. Veremos cómo se definen y utilizan las uniones dentro de un programa en C.

11.1. DEFINICIÓN DE UNA ESTRUCTURA

La declaración de estructuras es algo más complicada que la declaración de arrays, ya que una estructura debe ser definida en términos de sus miembros individuales. En general, la composición de una estructura puede ser definida como

```
struct marca {
      miembro 1;
      miembro 2;
      . . . . .
      miembro m;
};
```

En esta declaración, `struct` es una palabra reservada requerida; *marca* es un nombre que identifica estructuras de este tipo (estructuras que tengan esta composición); y *miembro 1*, *miembro 2*, . . . , *miembro m* son declaraciones de miembros individuales. (*Nota:* No existe distinción formal entre *definición* de estructura y *declaración* de estructura; ambos términos son intercambiables.)

Los miembros individuales pueden ser variables ordinarias, punteros, arrays u otras estructuras. Los nombres de los miembros dentro de una estructura particular deben ser todos diferentes, pero el nombre de un miembro puede ser el mismo que el de una variable definida fuera de la estructura. Sin embargo, no se puede asignar un tipo de almacenamiento a un miembro individual, ni tampoco puede inicializarse dentro de la declaración del tipo de la estructura.

Una vez que la composición de la estructura ha sido definida, las variables individuales de este tipo de estructura pueden declararse como sigue:

```
tipo-almacenamiento struct
marca variable 1, variable 2, ..., variable n;
```

donde *tipo-almacenamiento* es un especificador opcional de tipo de almacenamiento, `struct` es la palabra reservada requerida, *marca* es el nombre que aparece en la declaración de estructura y *variable 1, variable 2, ..., variable n* son variables de estructura del tipo *marca*.

EJEMPLO 11.1. A continuación se muestra una declaración típica de estructura.

```
struct cuenta {
    int num_cuenta;
    char tipo_cuenta;
    char nombre[80];
    float saldo;
};
```

Esta estructura se llama `cuenta` (la marca es `cuenta`). Contiene cuatro miembros: un entero (`num_cuenta`), un carácter (`tipo_cuenta`), un array de 80 caracteres (`nombre[80]`) y una cantidad en coma flotante (`saldo`). La Figura 11.1 ilustra esquemáticamente la composición de esta estructura.

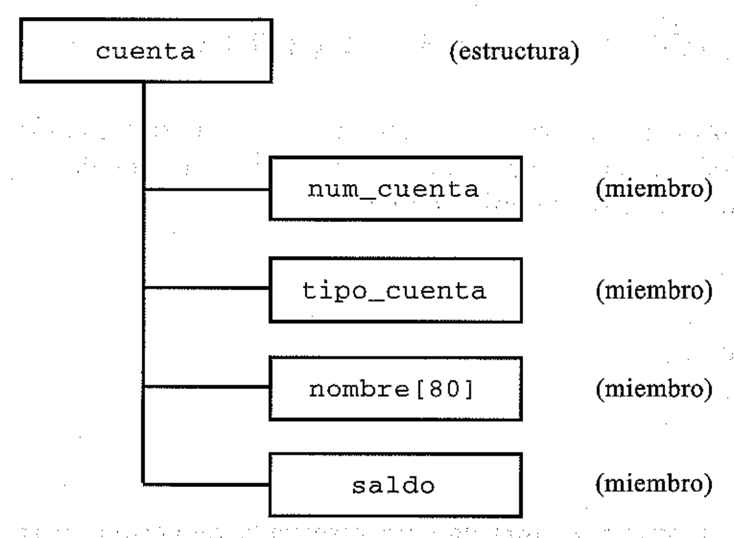


Figura 11.1.

Ahora podemos declarar las variables de estructura `antiguocliente` y `nuevocliente` como sigue:

```
struct cuenta antiguocliente, nuevocliente;
```

Así, `antiguocliente` y `nuevocliente` son variables de tipo `cuenta`. En otras palabras, `antiguocliente` y `nuevocliente` son variables de estructura cuya composición se identifica mediante la marca `cuenta`.

Es posible combinar la declaración de la composición de la estructura con la de las variables de estructura, como se muestra a continuación.

```
tipo-almacenamiento struct marca {
      miembro 1;
      miembro 2;
      . . . . .
      miembro m;
} variable 1, variable 2, . . ., variable n;
```

La *marca* es opcional en esta situación.

EJEMPLO 11.2. La siguiente declaración es equivalente a las dos declaraciones presentadas en el ejemplo anterior.

```
struct cuenta {
     int num_cuenta;
     char tipo_cuenta;
     char nombre[80];
     float saldo;
} antiguocliente, nuevocliente;
```

Así `antiguocliente` y `nuevocliente` son estructuras del tipo `cuenta`.

Como la declaración de variables se combina ahora con la declaración del tipo de la estructura, la marca (es decir, `cuenta`) no necesita ser incluida. Por tanto, la declaración puede ser escrita como

```
struct {
     int num_cuenta;
     char tipo_cuenta;
     char nombre[80];
     float saldo;
} antiguocliente, nuevocliente;
```

Una variable de estructura puede ser definida como miembro de otra estructura. En tales situaciones, la declaración de la estructura interna debe aparecer antes que la declaración de la estructura externa.

EJEMPLO 11.3. Un programa en C contiene las siguientes declaraciones de estructuras:

```
struct fecha {
     int mes;
     int dia;
     int anio;
};
```

```
struct cuenta {
     int num_cuenta;
     char tipo_cuenta;
     char nombre[80];
     float saldo;
     struct fecha ultimopago;
} antiguocliente, nuevocliente;
```

La segunda estructura (cuenta) contiene ahora otra estructura (fecha) como uno de sus miembros. Observe que la declaración de fecha precede a la declaración de cuenta. La Figura 11.2 muestra de modo esquemático la composición de cuenta.

A los miembros de una variable de estructura se le pueden asignar valores iniciales de la misma forma que a los arrays. Los valores iniciales deben aparecer en el orden en que serán asignados a sus correspondientes miembros de la estructura, encerrados entre llaves y separados por comas. La forma general es

```
tipo-almacenamiento struct marca variable =
        {valor 1, valor 2, . . ., valor m};
```

donde *valor 1* refiere al valor del primer miembro, *valor 2* al valor del segundo, y así sucesivamente. Una variable de estructura, al igual que un array, solamente puede ser inicializado si su tipo de almacenamiento es extern o static.

EJEMPLO 11.4. Este ejemplo ilustra la asignación de valores iniciales a los miembros de una variable estructura.

```
struct fecha {
     int mes;
     int dia;
     int anio;
};

struct cuenta {
     int num_cuenta;
     char tipo_cuenta;
     char nombre[80];
     float saldo;
     struct fecha ultimopago;
};

static struct cuenta cliente = {12345, 'R', "José R. García", 586.30,
                                5, 24, 90};
```

Así, cliente es una variable de estructura estática del tipo cuenta, cuyos miembros tienen asignados valores iniciales. El primer miembro (num_cuenta) tiene asignado el valor entero 12345, el segundo miembro (tipo_cuenta) tiene asignado el carácter 'R', el tercer miembro (nombre[80]) tiene asignada la cadena "José R. García" y el cuarto miembro (saldo) tiene asignado el valor en coma flotante 586.30. El último miembro es una estructura que contiene tres miembros enteros (mes, dia y anio). Por tanto, el último miembro de cliente tiene asignados los valores enteros 5, 24 y 90.

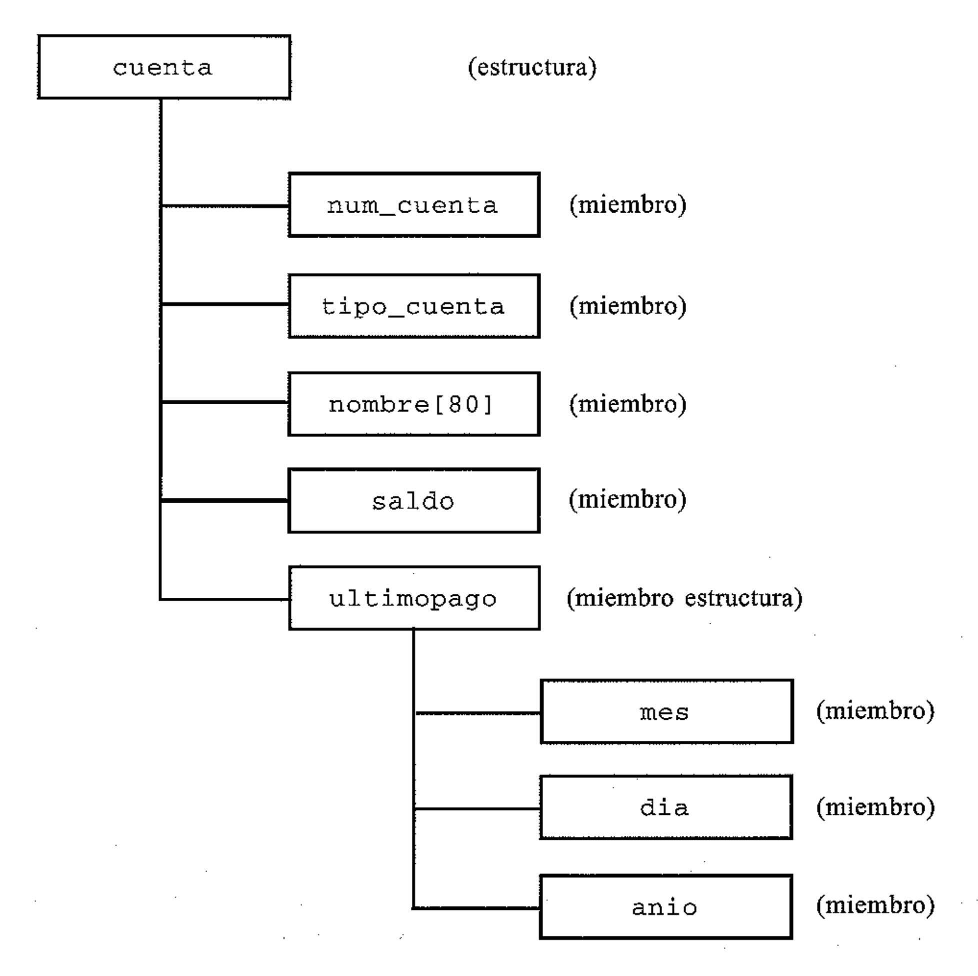


Figura 11.2.

También es posible definir un array de estructuras; esto es, un array en el que cada elemento sea una estructura. El procedimiento se ilustra en el siguiente ejemplo.

EJEMPLO 11.5. Un programa en C contiene las siguientes declaraciones de estructuras.

```
struct fecha {
     int mes;
     int dia;
     int anio;
};

struct cuenta {
     int num_cuenta;
     char tipo_cuenta;
     char nombre[80];
     float saldo;
     struct fecha ultimopago;
} cliente[100];
```

En esta declaración `cliente` es un array de `100` estructuras. Cada elemento de `cliente` es una estructura del tipo `cuenta` (cada elemento de `cliente` representa un registro de cliente individual).

Observe que cada estructura del tipo `cuenta` incluye un array `(nombre[80])` y otra estructura `(fecha)` como miembros. Así tenemos un array y una estructura incluidas dentro de otra estructura, que es a su vez un elemento de un array.

Por supuesto, también se puede definir `cliente` en una declaración separada, como se muestra a continuación.

```
struct fecha {
     int mes;
     int dia;
     int anio;
};

struct cuenta {
     int num_cuenta;
     char tipo_cuenta;
     char nombre[80];
     float saldo;
     struct fecha ultimopago;
};

struct cuenta cliente[100];
```

Un array de estructuras puede tener asignados valores iniciales como cualquier otro array. Recordar que cada elemento del array es una estructura a la que debe asignarse su correspondiente conjunto de valores iniciales, como se ilustra a continuación.

EJEMPLO 11.6. Un programa en C contiene las siguientes declaraciones:

```
struct fecha {
     char nombre[80];
     int mes;
     int dia;
     int anio;
};

static struct fecha nacimiento[] = { "Ana", 12, 30, 73,
                                    "José", 5, 13, 66,
                                    "Lidia", 7, 15, 72,
                                    "Raúl", 11, 29, 70,
                                    "Rosa", 2, 4, 77,
                                    "Sara", 12, 29, 63,
                                    "Zoraida", 4, 12, 69};
```

En este ejemplo `nacimiento` es un array de estructuras cuyo tamaño está sin especificar. Los valores iniciales definirán el tamaño del array y la cantidad de memoria necesaria para almacenar el array.

Observe que cada fila en la declaración de la variable contiene cuatro constantes. Estas constantes representan los valores iniciales, esto es, el nombre, mes, día y año, para un elemento del array. Como hay siete filas (siete conjuntos de constantes), el array contendrá siete elementos numerados de `0` a `6`.

Algunos programadores prefieren incluir cada conjunto de constantes dentro de un par de llaves para separar más claramente los elementos individuales del array. Esto está permitido. Así, la declaración del array se puede escribir

```
static struct fecha nacimiento[] = {
                                      {"Ana", 12, 30, 73},
                                      {"José", 5, 13, 66},
                                      {"Lidia", 7, 15, 72},
                                      {"Raúl", 11, 29, 70},
                                      {"Rosa", 2, 4, 77},
                                      {"Sara", 12, 29, 63},
                                      {"Zoraida", 4, 12, 69}
                                    };
```

Recordar que cada estructura, con respecto a las definiciones de miembros, es una entidad autónoma. Así, el mismo nombre de miembro puede usarse en diferentes estructuras para representar diferentes datos. En otras palabras, el ámbito de un nombre de miembro está confinado a la estructura particular dentro de la cual ha sido definido.

EJEMPLO 11.7. Dos estructuras distintas, llamadas `primera` y `segunda`, son declaradas a continuación.

```
struct primera {
     float a;
     int b;
     char c;
};

struct segunda {
     char a;
     float b, c;
};
```

Observe que los nombres de los miembros individuales `a`, `b` y `c` aparecen en ambas declaraciones de estructura, pero los tipos de datos asociados son diferentes. Así, `a` representa una cantidad en coma flotante en `primera` y un carácter en `segunda`. Análogamente, `b` representa un entero en `primera` y una cantidad en coma flotante en `segunda`, mientras que `c` representa un carácter en `primera` y una cantidad en coma flotante en `segunda`. Esta duplicación de nombres de miembros está permitida, ya que el ámbito de cada conjunto de definiciones de miembros está confinado a su respectiva estructura. Dentro de cada estructura los nombres de miembros son distintos, como se requiere.

11.2. PROCESAMIENTO DE UNA ESTRUCTURA

Los miembros de una estructura se procesan generalmente de modo individual, como entidades separadas. Por tanto, tenemos que ser capaces de acceder a los miembros individuales de la estructura. Un miembro de una estructura puede ser accedido escribiendo

`variable.miembro`

donde *variable* refiere el nombre de una variable de tipo estructura y *miembro* el nombre de un miembro dentro de la estructura. Observe el punto (.) que separa el nombre de la variable del nombre del miembro. Este punto es un operador; es un miembro del grupo de mayor prioridad, y su asociatividad es de izquierda a derecha (ver Apéndice C).

EJEMPLO 11.8. Considerar las siguientes declaraciones de estructuras:

```
struct fecha {
     int mes;
     int dia;
     int anio;
};

struct cuenta {
     int num_cuenta;
     char tipo_cuenta;
     char nombre[80];
     float saldo;
     struct fecha ultimopago;
} cliente;
```

En este ejemplo `cliente` es una estructura del tipo `cuenta`. Si queremos acceder al número de cuenta del cliente, debemos escribir

```
cliente.num_cuenta
```

Análogamente, el nombre y el saldo del cliente pueden ser accedidos escribiendo

```
cliente.nombre
```

y

```
cliente.saldo
```

Como el operador punto pertenece al grupo de mayor precedencia, tendrá precedencia sobre los operadores unarios, así como sobre los operadores aritméticos, relacionales, lógicos y de asignación. Por ejemplo, una expresión de la forma `++`*variable*`.`*miembro* es equivalente a `++(`*variable*`.`*miembro*`)`; es decir, el operador ++ se aplicará sobre el miembro de la estructura y no sobre la estructura completa. Análogamente, la expresión `&`*variable*`.`*miembro* es equivalente a `&(`*variable*`.`*miembro*`)`; así la expresión accede a la dirección del miembro de la estructura y no a la dirección de comienzo de la variable estructura.

EJEMPLO 11.9. Consideremos las declaraciones de estructura dadas en el Ejemplo 11.8.

```
struct fecha {
     int mes;
     int dia;
     int anio;
};
```

```
struct cuenta {
     int num_cuenta;
     char tipo_cuenta;
     char nombre[80];
     float saldo;
     struct fecha ultimopago;
} cliente;
```

Varias expresiones que involucran a la variable de estructura `cliente` y a sus miembros son dadas a continuación.

Expresión	*Interpretación*
`++cliente.saldo`	Incrementa el valor de `cliente.saldo`
`cliente.saldo++`	Incrementa el valor de `cliente.saldo` después de acceder a su valor
`--cliente.num_cuenta`	Decrementa el valor de `cliente.num_cuenta`
`&cliente`	Accede a la dirección de comienzo de `cliente`
`&cliente.num_cuenta`	Accede a la dirección de `cliente.num_cuenta`

Se pueden escribir expresiones más complejas usando repetidamente el operador punto. Por ejemplo, si un miembro de una estructura es a su vez otra estructura, entonces el miembro de la estructura más interna puede ser accedido escribiendo

variable.miembro.submiembro

donde *miembro* refiere el nombre del miembro dentro de la estructura externa y *submiembro* el nombre del miembro dentro de la estructura interna. Análogamente, si un miembro de una estructura es un array, entonces un elemento individual del array puede ser accedido escribiendo

variable.miembro[*expresión*]

donde *expresión* es un valor no negativo que indica el elemento del array.

EJEMPLO 11.10. Consideremos de nuevo la declaración de estructura presentada en el Ejemplo 11.8.

```
struct fecha {
     int mes;
     int dia;
     int anio;
};

struct cuenta {
     int num_cuenta;
     char tipo_cuenta;
     char nombre[80];
     float saldo;
     struct fecha ultimopago;
} cliente;
```

El último miembro de `cliente` es `cliente.ultimopago`, que es una estructura de tipo `fecha`. Por tanto, para acceder al mes del último pago debemos escribir

```
cliente.ultimopago.mes
```

Además, este valor puede incrementarse escribiendo

```
++cliente.ultimopago.mes
```

Análogamente, el tercer miembro de `cliente` es el array de caracteres `cliente.nombre`. El tercer carácter dentro de este array puede ser accedido escribiendo

```
cliente.nombre[2]
```

La dirección de este carácter puede obtenerse como

```
&cliente.nombre[2]
```

El uso del operador punto puede extenderse a arrays de estructuras escribiendo

array[*expresión*].*miembro*

donde *array* refiere el nombre del array y *array*[*expresión*].*miembro* es un elemento individual del array (una variable de estructura). Por tanto, *array*[*expresión*] referirá un miembro específico dentro de una estructura particular.

EJEMPLO 11.11. Consideremos la siguiente declaración de estructura que se presentó originalmente en el Ejemplo 11.5.

```
struct fecha {
     int mes;
     int dia;
     int anio;
};

struct cuenta {
     int num_cuenta;
     char tipo_cuenta;
     char nombre[80];
     float saldo;
     struct fecha ultimopago;
} cliente[100];
```

En este ejemplo `cliente` es un array que puede contener hasta `100` elementos. Cada elemento es una estructura del tipo `cuenta`. Así, si queremos acceder al número de cuenta del cliente `14` (`cliente[13]`, ya que el índice empieza en `0`), debemos escribir `cliente[13].num_cuenta`. Análogamente, se puede acceder al saldo de este cliente escribiendo `cliente[13].saldo`.

Se puede acceder al nombre del cliente 14 escribiendo `cliente[13].nombre`. Además, podemos acceder a caracteres individuales dentro del nombre especificando un índice. Por ejemplo, el octavo carácter dentro del nombre puede ser accedido escribiendo `cliente[13].nombre[7]`. De forma similar podemos acceder al mes, día y año del último pago del cliente 14 especificando los miembros individuales de `cliente[13].ultimopago`, que son `cliente[13].ultimopago.mes`, `cliente[13].ultimopago.dia` y `cliente[13].ultimopago.anio`. Además, la expresión `++cliente[13].ultimopago.dia` produce un incremento en el valor del día.

Los miembros de una estructura pueden procesarse de la misma manera que las variables ordinarias de este mismo tipo. Los miembros de estructura unievaluados pueden aparecer en expresiones, pueden pasarse y ser devueltos por funciones como si se tratara de variables ordinarias unievaluadas. Los miembros complejos de estructuras se procesan de la misma manera que los datos ordinarios del mismo tipo. Por ejemplo, un miembro de estructura que es un array puede procesarse del mismo modo que un array ordinario, y con las mismas restricciones. Análogamente, un miembro de estructura que sea a su vez otra estructura puede ser procesado sobre la base de miembro por miembro (aquí los miembros hacen referencia a los de la estructura anidada), lo mismo que para cualquier otra estructura.

EJEMPLO 11.12. A continuación se muestran varios grupos de instrucciones que acceden a miembros individuales de estructura. Todos los miembros de la estructura se definen conforme a la declaración dada en el Ejemplo 11.8.

```
cliente.saldo = 0;

cliente.saldo -= pagos;

cliente.ultimopago.mes = 12;

printf("Nombre: %s\n", cliente.nombre);

if (cliente.tipo_cuenta == 'P')
   printf("Cuenta preferente nº: %d\n", cliente.num_cuenta);
else
   printf("Cuenta regular nº: %d\n", cliente.num_cuenta);
```

La primera instrucción asigna el valor cero a `cliente.saldo`, mientras que la segunda produce el decremento del valor de `cliente.saldo` por el valor de `pagos`. La tercera instrucción hace que el valor 12 sea asignado a `cliente.ultimopago.mes`. Observe que `cliente.ultimopago.mes` es un miembro de la estructura anidada `cliente.ultimopago`.

La cuarta instrucción pasa el array `cliente.nombre` a la función `printf`, mostrando el nombre del cliente. Finalmente, el último ejemplo ilustra el uso de miembros de la estructura dentro de una instrucción `if - else`. También vemos una situación en que el miembro de la estructura `cliente.num_cuenta` se pasa a una función como argumento.

En algunas de las versiones más antiguas de C, las estructuras debían ser procesadas miembro por miembro. Con esta restricción, la única operación permisible con la estructura completa es tomar su dirección (más sobre esto más adelante). Sin embargo, la mayoría de las nuevas versiones permiten asignar una estructura completa a otra siempre que las estructuras tengan la misma composición. Esta característica se incluye en el nuevo estándar ANSI.

EJEMPLO 11.13. Supongamos que `anteriorcliente` y `nuevocliente` son estructuras con la misma composición; esto es,

```
struct fecha {
     int mes;
     int dia;
     int anio;
};

struct cuenta {
     int num_cuenta;
     char tipo_cuenta;
     char nombre[80];
     float saldo;
     struct fecha ultimopago;
} anteriorcliente, nuevocliente;
```

como se declaró en el Ejemplo 11.8. Supongamos que a todos los miembros de `anteriorcliente` se le han asignado valores individuales. En la mayoría de las nuevas versiones de C es posible copiar estos valores a `nuevocliente` simplemente escribiendo

```
nuevocliente = anteriorcliente;
```

Por otra parte, alguna de las antiguas versiones de C pueden requerir que los valores sean copiados individualmente, miembro a miembro; por ejemplo,

```
nuevocliente.num_cuenta = anteriorcliente.num_cuenta;
nuevocliente.tipo_cuenta = anteriorcliente.tipo_cuenta;

. . . . .

nuevocliente.ultimopago.anio = anteriorcliente.ultimopago.anio;
```

También es posible pasar estructuras completas a y desde funciones, pero la forma de hacerlo varía de una versión de C a otra. Las versiones antiguas de C sólo permiten pasar punteros, mientras que el estándar ANSI permite pasar las estructuras completas. Discutiremos esto posteriormente en la sección 11.5. Sin embargo, antes de tratar la relación entre estructuras y punteros y los métodos de pasar estructuras a funciones, consideremos un ejemplo más sencillo que involucre el procesamiento de miembros de la estructura.

EJEMPLO 11.14. Actualización de registros de clientes. Para ilustrar más completamente cómo se procesan los miembros individuales de una estructura, consideremos un sistema de facturación de clientes muy simple. En este sistema los registros de clientes se almacenan en un array de estructuras. Cada registro será almacenado como una estructura individual (como un elemento del array), conteniendo el nombre del cliente, dirección con calle, ciudad y estado, número de cuenta, estado de la cuenta (al día, atrasada o delincuente), saldo previo, nuevo saldo y fecha de pago.

La estrategia será introducir cada registro en la computadora, y actualizarlo tan pronto como sea introducido, para reflejar los pagos actuales. Entonces se mostrarán todos los registros actualizados,

con el estado actual de cada cuenta. El estado de la cuenta se basará en el último pago realizado y el saldo previo del cliente.

A continuación se muestran las declaraciones de estructuras.

```
struct fecha {
     int mes;
     int dia;
     int anio;
};

struct cuenta {
     char nombre[80];
     char calle[80];
     char ciudad[80];
     int num_cuenta;
     int tipo_cuenta;
     float anteriorsaldo;
     float nuevosaldo;
     float pago;
     struct fecha ultimopago;
} cliente[100];
```

Observe que `cliente` es un array de `100` estructuras. Así, cada elemento del array (cada estructura) representará un registro de cliente. Cada estructura incluye tres miembros que son arrays de caracteres (`nombre`, `calle` y `ciudad`) y un miembro que es otra estructura (`ultimopago`).

El estado de cada cuenta será determinado de la siguiente manera:

1. Si el pago actual es mayor que cero pero menor que el diez por ciento del saldo previamente al descubierto, la cuenta será atrasada.
2. Si hay un saldo al descubierto y el pago actual es cero, la cuenta será delincuente.
3. En otro caso, la cuenta estará al día.

La estrategia global será como sigue:

1. Especificar el número de cuentas de clientes (número de estructuras) a procesar.
2. Para cada cliente, leer los siguientes elementos:

   a) nombre
   b) calle
   c) ciudad
   d) número de cuenta
   e) saldo anterior
   f) pago actual
   g) fecha de pago

3. Cada vez que se lee un registro en la computadora, actualizarlo de la siguiente manera:

   a) Comparar el pago actual con el saldo anterior y determinar el estado apropiado de la cuenta.
   b) Calcular el nuevo saldo restando el pago actual del saldo anterior (un saldo negativo indicaría un crédito).

4. Una vez que todos los registros han sido introducidos y procesados, escribir la siguiente información para cada registro:

   a) nombre
   b) número de cuenta
   c) calle
   d) ciudad

e) saldo anterior
f) pago actual
g) nuevo saldo
h) estado de la cuenta

Escribamos ahora el programa de forma modular, con una función para introducir y actualizar cada registro y otra función para mostrar los datos actualizados. Idealmente deberíamos pasar cada registro de cliente (cada elemento del array) a cada una de estas funciones. Sin embargo, como cada registro de cliente es una estructura y todavía no hemos discutido cómo se pasa una estructura a o desde una función, definiremos el array de estructuras como un array externo. Esto nos permitirá acceder a los elementos del array, y a los miembros individuales de la estructura, directamente desde todas las funciones.

Los módulos individuales del programa son directos, pero se necesita algún cuidado para leer los miembros individuales de la estructura dentro de la computadora. Aquí está el programa completo.

```
/* actualizar una serie de cuentas de clientes (sistema de factu-
   ración simplificado) */
/* mantener las cuentas de clientes como un array externo de es-
   tructuras */

#include <stdio.h>

void leerentrada(int i);
void escribirsalida(int i);

struct fecha {
   int mes;
   int dia;
   int anio;
};

struct cuenta {
   char nombre[80];
   char calle[80];
   char ciudad[80];
   int num_cuenta;              /* (entero positivo) */
   int tipo_cuenta;             /* A (Al día), R (atrasada) o D (de-
                                   lincuente) */
   float anteriorsaldo;         /* (cantidad no negativa) */
   float nuevosaldo;            /* (cantidad no negativa) */
   float pago;                  /* (cantidad no negativa) */
   struct fecha ultimopago;
} cliente[100];                 /* mantener hasta 100 clientes */

main()
{
   int i, n;

   printf("SISTEMA DE FACTURACION DE CLIENTES\n\n");
   printf("¿Cuantos clientes hay? ");
   scanf("%d", &n);

   for (i = 0; i < n; ++i) {
       leerentrada(i);
```

```
        /* determinar el estado de la cuenta */

        if (cliente[i].pago > 0)
            cliente[i].tipo_cuenta =
                (cliente[i].pago < 0.1 * cliente[i].anteriorsaldo) ?
              'R'                                 : 'A';
        else
            cliente[i].tipo_cuenta =
                (cliente[i].anteriorsaldo > 0) ? 'D' : 'A';

        /* ajustar el saldo de la cuenta */

         cliente[i].nuevosaldo  =  cliente[i].anteriorsaldo
                                  - cliente[i].pago;
    }

    for (i = 0; i < n; ++i)
         escribirsalida(i);
}

void leerentrada(int i)

/* leer datos de entrada y actualizar el registro para cada cliente */

{
   printf("\nCliente nº %d\n", i + 1);
   printf("    Nombre: ");
   scanf(" %[^\n]", cliente[i].nombre);
   printf("    Calle: ");
   scanf(" %[^\n]", cliente[i].calle);
   printf("    Ciudad: ");
   scanf(" %[^\n]", cliente[i].ciudad);
   printf("    Número de cuenta: ");
   scanf("%d", &cliente[i].num_cuenta);
   printf("    Saldo anterior: ");
   scanf("%f", &cliente[i].anteriorsaldo);
   printf("    Pago actual: ");
   scanf("%f", &cliente[i].pago);
   printf("    Fecha de pago (mm/dd/aaaa): ");
   scanf("%d/%d/%d",  &cliente[i].ultimopago.mes,
                      &cliente[i].ultimopago.dia,
                      &cliente[i].ultimopago.anio);
   return;
}

void escribirsalida(int i)

/* escribir la información actual para cada cliente */

{
   printf("\nNombre: %s", cliente[i].nombre);
   printf("       Número de cuenta: %d\n", cliente[i].num_cuenta);
```

```
    printf("Calle:   %s\n", cliente[i].calle);
    printf("Ciudad:  %s\n\n", cliente[i].ciudad);
    printf("Saldo anterior: %7.2f", cliente[i].anteriorsaldo);
    printf("     Pago actual: %7.2f", cliente[i].pago);
    printf("     Nuevo saldo: %7.2f\n\n", cliente[i].nuevosaldo);
    printf("Estado de la cuenta: ");

    switch (cliente[i].tipo_cuenta) {
    case 'A':
                    printf("AL DIA\n\n");
                    break;
    case 'R':
                    printf("ATRASADA\n\n");
                    break;
    case 'D':
                    printf("DELINCUENTE\n\n");
                    break;
    }
    return;
}
```

Supongamos ahora que el programa se utiliza para procesar cuatro registros de clientes ficticios. A continuación se muestra la entrada interactiva, con las respuestas de usuario subrayadas.

```
SISTEMA DE FACTURACION DE CLIENTES

"¿Cuantos clientes hay? 4

Cliente nº 1
     Nombre: Steve Johnson
     Calle: 123 Mountainview Drive
     Ciudad: Denver, CO
     Número de cuenta: 4208
     Saldo anterior: 247.88
     Pago actual: 25.00
     Fecha de pago (mm/dd/aaaa): 6/14/1998

Cliente nº 2
     Nombre: Susan Richards
     Calle: 4383 Alligator Blvd
     Ciudad: Fort Lauderdale, FL
     Número de cuenta: 2219
     Saldo anterior: 135.00
     Pago actual: 135.00
     Fecha de pago (mm/dd/aaaa): 8/10/2000

Cliente nº 3
     Nombre: Martin Peterson
     Calle: 1787 Pacific Parkway
     Ciudad: San Diego, CA
```

```
    Número de cuenta: 8452
    Saldo anterior: 387.42
    Pago actual: 35.00
    Fecha de pago (mm/dd/aaaa): 9/22/1999

Cliente nº 4
    Nombre: Phyllis Smith
    Calle: 1000 Greay White Way
    Ciudad: New York, NY
    Número de cuenta: 711
    Saldo anterior: 260.00
    Pago actual: 0
    Fecha de pago (mm/dd/aaaa): 11/27/2001
```

El programa generará los siguientes datos de salida:

```
Nombre: Steve Johnson     Número de cuenta: 4208
Calle:  123 Mountain Drive
Ciudad: Denver, CO

Saldo anterior:  247.88   Pago actual:  25.00    Nuevo saldo:  222.88

Estado de la cuenta: AL DIA

Nombre: Susan Richards    Número de cuenta: 2219
Calle:  4383 Alligator Blvd
Ciudad: Fort Lauderdale, FL

Saldo anterior:  135.00   Pago actual:  135.00  Nuevo saldo:  0.00

Estado de la cuenta: AL DIA

Nombre: Martin Peterson   Número de cuenta: 8452
Calle:  1787 Pacific Parkway
Ciudad: San Diego, CA

Saldo anterior:  387.42   Pago actual:  35.00   Nuevo saldo:  352.42

Estado de la cuenta: ATRASADA

Nombre: Phyllis Smith     Número de cuenta: 711
Calle:  1000 Great White Way
Ciudad: New York, NY

Saldo anterior:  260.00   Pago actual:  0.00    Nuevo saldo:  260.00

Estado de la cuenta: DELINCUENTE
```

Debe quedar claro que este ejemplo es irreal desde el punto de vista práctico, por dos razones. Primero, el array de estructuras (`cliente`) se define como externo para todas las funciones del programa.

Sería preferible declarar cliente dentro de main y entonces pasarlo a o desde leerentrada o escribirsalida según se necesite. Aprenderemos cómo se hace esto en la sección 11.5.

Un problema más serio es el hecho de que el sistema de facturación de clientes real almacenará los registros de clientes en un archivo de datos de un dispositivo de memoria auxiliar, tal como un disco duro o una cinta magnética. Para actualizar el registro deberíamos acceder al registro del archivo de datos, cambiar los datos necesarios y entonces escribir el registro modificado en el archivo de datos. La utilización de archivos de datos se trata en el Capítulo 12. Como el presente ejemplo no hace uso de archivos de datos, debemos reintroducir todos los registros de clientes cada vez que se ejecute el programa. Está poco logrado, pero provee un ejemplo simple para ilustrar la manera en que se pueden procesar las estructuras miembro a miembro.

A veces es útil el determinar el número de bytes requeridos por un array o una estructura. Esta información puede obtenerse mediante el uso del operador sizeof, discutido originalmente en la sección 3.2. Por ejemplo, el tamaño de la estructura puede ser determinado escribiendo sizeof *variable* o sizeof (struct *marca*).

EJEMPLO 11.15. A continuación se muestra un programa elemental en C.

```
#include <stdio.h>

main()   /* determinar el tamaño de una estructura */
{
   struct fecha {
      int mes;
      int dia;
      int anio;
   };

   struct cuenta {
      int num_cuenta;
      char tipo_cuenta;
      char nombre[80];
      float saldo;
      struct fecha ultimopago;
   } cliente;

   printf("%d\n", sizeof cliente);
   printf("%d", sizeof (struct cuenta));
}
```

Este programa hace uso del operador sizeof para determinar el número de bytes asociados con la variable de estructura cliente (o equivalentemente la estructura cuenta). Las dos instrucciones printf ilustran diferentes modos de usar el operador sizeof. Ambas instrucciones printf producen la misma salida.

La ejecución del programa generará la siguiente salida:

```
93
93
```

Así, la variable de estructura `cliente` (o la estructura `cuenta`) ocupará `93` bytes. Este valor se obtiene como sigue:

Miembro de la estructura	*Número de bytes*
`num_cuenta`	2
`tipo_cuenta`	1
`nombre`	80
`saldo`	4
`ultimopago`	6
Total	93

Algunos compiladores pueden asignar dos bytes a `tipo_cuenta` para mantener un número par de bytes. De ahí que el total de bytes pueda ser `94` en vez de `93`.

11.3. TIPOS DE DATOS DEFINIDOS POR EL USUARIO (`typedef`)

La característica `typedef` permite a los usuarios definir nuevos tipos de datos que sean equivalentes a los tipos de datos existentes. Una vez que el tipo de datos definido por el usuario ha sido establecido, entonces las nuevas variables, arrays, estructuras, etc., pueden ser declaradas en términos de este nuevo tipo de datos.

En términos generales, un nuevo tipo de datos se define como

```
typedef tipo nuevo-tipo;
```

donde *`tipo`* refiere un tipo de datos existente (bien un tipo de datos estándar o bien un tipo de datos previamente definido por el usuario) y *`nuevo-tipo`* el nuevo tipo de datos definido por el usuario. Sin embargo, debe quedar claro que el nuevo tipo será nuevo solo en el nombre. En realidad, este nuevo tipo de datos no será fundamentalmente diferente de los tipos de datos estándar.

EJEMPLO 11.16. Aquí tenemos una declaración simple que involucra el uso de `typedef`.

```
typedef int edad;
```

En esta declaración `edad` es un tipo de datos definido por el usuario equivalente al tipo `int`. De aquí, la declaración de variable

```
edad  varon, hembra;
```

es equivalente a escribir

```
int  varon, hembra;
```

En otras palabras, `varon` y `hembra` se consideran variables de tipo `edad`, pero son realmente variables de tipo entero.

Análogamente, las declaraciones

```
typedef   float   altura[100];
altura    hombres, mujeres;
```

define `altura` como un array de `100` elementos en coma flotante.

Por tanto, `hombres` y `mujeres` son arrays de `100` elementos en coma flotante. Otra forma de expresar esto es

```
typedef   float   altura;
altura    hombres[100],  mujeres[100];
```

si bien la declaración anterior es algo más sencilla.

La característica `typedef` es particularmente útil cuando se definen estructuras, ya que se elimina la necesidad de escribir repetidamente `struct` *`marca`* cuando se referencia una estructura. Como resultado, la estructura puede ser referenciada más concisamente. Además, el nombre dado al tipo de estructura definido por el usuario sugiere a menudo el propósito de la estructura dentro de un programa.

En términos generales, el tipo de estructura definida por el usuario se puede escribir como

```
typedef  struct  {
    miembro 1;
    miembro 2;
    . . . . .
    miembro m;
} nuevo-tipo;
```

donde *`nuevo-tipo`* es el tipo de estructura definida por el usuario. Las variables de estructura pueden definirse en términos del nuevo tipo de datos.

EJEMPLO 11.17. Las siguientes declaraciones son comparables a las declaraciones de estructura presentadas en el Ejemplo 11.1 y 11.2. Sin embargo, ahora introducimos un tipo de datos definido por el usuario para describir la estructura.

```
typedef  struct  {
    int num_cuenta;
    char  tipo_cuenta;
    char  nombre[80];
    float  saldo;
} registro;

registro  anteriorcliente,  nuevocliente;
```

La primera declaración define `registro` como un tipo de datos definido por el usuario. La segunda decla-ración define `anteriorcliente` y `nuevocliente` como variables de estructura de tipo `registro`.

La característica `typedef` puede ser utilizada repetidamente para definir un tipo de datos en términos de otros tipos de datos definidos por el usuario.

EJEMPLO 11.18. A continuación se muestran algunas variaciones de las declaraciones de estructuras presentadas en el Ejemplo 11.5.

```
typedef struct {
    int mes;
    int dia;
    int anio;
} fecha;

typedef struct {
    int num_cuenta;
    char tipo_cuenta;
    char nombre[80];
    float saldo;
    fecha ultimopago;
} registro;

registro cliente[100];
```

En este ejemplo `fecha` y `registro` son tipos de estructuras definidas por el usuario y `cliente` es un array de `100` elementos cuyos elementos son estructuras del tipo `registro`. (Recordar que `fecha` era una marca en vez del tipo de dato real en el Ejemplo 11.5.) Los miembros individuales dentro del *i*-ésimo elemento de `cliente` pueden ser escritos como `cliente[i].num_cuenta`, `cliente[i].nombre`, `cliente[i].ultimopago.mes`, etc., como antes.

Por supuesto, hay variaciones sobre este tema. Así, una declaración alternativa puede escribirse como

```
typedef struct {
    int mes;
    int dia;
    int anio;
} fecha;

typedef struct {
    int num_cuenta;
    char tipo_cuenta;
    char nombre[80];
    float saldo;
    fecha ultimopago;
} registro[100];

registro cliente;
```

o simplemente

```
typedef struct {
    int mes;
    int dia;
    int anio;
} fecha;
```

```
struct  {
    int num_cuenta;
    char tipo_cuenta;
    char nombre[80];
    float saldo;
    fecha ultimopago;
} cliente[100];
```

Estos tres conjuntos de declaraciones son equivalentes.

11.4. ESTRUCTURAS Y PUNTEROS

Podemos acceder a la dirección de comienzo de una estructura de la misma manera que cualquier otra dirección, mediante el uso del operador dirección (&). Así, si `variable` representa un tipo de variable de estructura, entonces `&variable` representa la dirección de comienzo de esa variable. Además, podemos declarar una variable puntero a una estructura escribiendo

```
tipo  *ptvar;
```

donde *tipo* es el tipo de datos que identifica la composición de la estructura y *ptvar* representa el nombre de la variable puntero. Podemos asignar la dirección de comienzo de la variable de estructura a este puntero escribiendo

```
ptvar = &variable;
```

EJEMPLO 11.19. Consideremos la siguiente declaración de estructura, que es una variación de la declaración presentada en el Ejemplo 11.1:

```
typedef  struct  {
    int num_cuenta;
    char tipo_cuenta;
    char nombre[80];
    float saldo;
} cuenta;

cuenta  cliente, *pc;
```

En este ejemplo `cliente` es una variable de estructura de tipo `cuenta` y `pc` un puntero cuyo objeto es una variable de estructura del tipo `cuenta`. Así, la dirección de comienzo de `cliente` puede ser asignada a `pc` escribiendo

```
pc = &cliente;
```

Las declaraciones de la variable y del puntero pueden combinarse con la declaración de la estructura escribiendo

```
struct {
    miembro 1;
```

```
    miembro 2;
    . . . . .
    miembro m;
} variable, *ptvar;
```

donde *variable* representa de nuevo una variable del tipo estructura y *ptvar* el nombre de una variable puntero.

EJEMPLO 11.20. La siguiente declaración es equivalente a las dos declaraciones presentadas en el ejemplo anterior.

```
struct {
    int num_cuenta;
    char tipo_cuenta;
    char nombre[80];
    float saldo;
} cliente, *pc;
```

La dirección de comienzo de `cliente` puede asignarse a `pc` escribiendo

```
pc = &cliente;
```

como en el ejemplo previo.

Podemos acceder a un miembro individual de una estructura en términos de su correspondiente variable puntero escribiendo

```
ptvar->miembro
```

donde *ptvar* refiere una variable puntero a estructura y el operador `->` es comparable al operador punto (`.`) discutido en la sección 11.2. Así, la expresión

```
ptvar->miembro
```

es equivalente a escribir

```
variable.nombre
```

donde *variable* es una variable de estructura, como se discutió en la sección 11.2. El operador `->` pertenece al grupo de mayor precedencia, como el operador punto (`.`). Su asociatividad es de izquierda a derecha (ver Apéndice C).

El operador `->` puede combinarse con el operador punto para acceder a un submiembro dentro de una estructura (para acceder a un miembro de una estructura que es a su vez miembro de otra estructura). Por tanto, un submiembro puede ser accedido escribiendo

```
ptvar->miembro.submiembro
```

Análogamente, el operador `->` puede usarse para acceder a un elemento de un array que es miembro de una estructura. Esto se realiza escribiendo

```
ptvar->miembro[expresión]
```

donde *expresión* es un entero no negativo que indica el elemento del array.

EJEMPLO 11.21. A continuación se muestra una variación de las declaraciones mostradas en el Ejemplo 11.8.

```
typedef  struct  {
    int mes;
    int dia;
    int anio;
} fecha;

struct  {
    int num_cuenta;
    char tipo_cuenta;
    char nombre[80];
    float saldo;
    fecha ultimopago;
} cliente, *pc = &cliente;
```

Observe que la variable puntero `pc` es inicializada asignándole la dirección de comienzo de la variable de estructura `cliente`. En otras palabras, `pc` apuntará a `cliente`.

Si queremos acceder al número de cuenta del cliente, podremos escribir cualquiera de las siguientes expresiones:

```
cliente.num_cuenta        pc->num_cuenta        (*pc).num_cuenta
```

En la última expresión se necesitan los paréntesis porque el operador punto tiene mayor precedencia que el operador indirección (`*`). Sin los paréntesis el compilador generará un error, porque `pc` (un puntero) no es directamente compatible con el operador punto.

Análogamente, podemos acceder al saldo del cliente escribiendo cualquiera de las siguientes expresiones:

```
cliente.saldo   pc->saldo       (*pc).saldo
```

y el mes del último pago puede ser accedido escribiendo cualquiera de las siguientes expresiones:

```
cliente.ultimopago.mes    pc->ultimopago.mes    (*pc).ultimopago.mes
```

Finalmente, el nombre del cliente puede ser accedido escribiendo cualquiera de las siguientes expre-siones:

```
cliente.nombre     pc->nombre     (*pc).nombre
```

Por tanto, el tercer carácter del nombre del cliente puede ser accedido escribiendo cualquiera de las siguientes expresiones (ver sección 10.4):

```
cliente.nombre[2]         pc->nombre[2]         (*pc).nombre[2]

*(cliente.nombre + 2)    pc->(nombre + 2)      *((*pc).nombre + 2)
```

Una estructura puede incluir también uno o más punteros como miembros. Así, si *`ptmiembro`* es a la vez un puntero y un miembro de *`variable`*, entonces *`*variable.ptmiembro`* accederá al valor al cual apunta *`ptmiembro`*. Análogamente, si *`ptvar`* es una variable puntero que apunta a una estructura y *`ptmiembro`* es un miembro de esa estructura, entonces *`*ptvar->ptmiembro`* accederá al valor al que apunta *`ptmiembro`*.

EJEMPLO 11.22. Consideremos el programa sencillo en C mostrado a continuación.

```
#include <stdio.h>

main()
{
   int n = 3333;
   char t = 'A';
   float b = 99.99;

   typedef struct {
      int mes;
      int dia;
      int anio;
   } fecha;

   struct {
      int *num_cuenta;
      char *tipo_cuenta;
      char *nombre;
      float *saldo;
      fecha ultimopago;
   } cliente, *pc = &cliente;

   cliente.num_cuenta = &n;
   cliente.tipo_cuenta = &t;
   cliente.nombre = "Lázaro";
   cliente.saldo = &b;

   printf("%d %c %s %.2f\n", *cliente.num_cuenta, *cliente.tipo_
                              cuenta, cliente.nombre, *cliente.saldo);
   printf("%d %c %s %.2f\n", *pc->num_cuenta, *pc->tipo_cuenta,
                              pc->nombre, *pc->saldo);
}
```

Dentro de la segunda estructura, los miembros `num_cuenta`, `tipo_cuenta` y `saldo` están escritos como punteros. Así, el valor al que apunta `num_cuenta` puede ser accedido escribiendo `*cliente.num_cuenta` o `*pc->num_cuenta`. Lo mismo es cierto para `tipo_cuenta` y `saldo`. Además, recordar que una cadena de caracteres puede ser directamente asignada a un puntero a carácter. Por tanto, si `nombre` apunta al principio de la cadena, entonces se puede acceder a la cadena escribiendo `cliente.nombre` o `pc->nombre`.

La ejecución de este programa genera las dos siguientes líneas de salida:

```
3333 A Lázaro 99.99

3333 A Lázaro 99.99
```

Como era de esperar, las dos líneas de salida son idénticas.

Como el operador -> es miembro del grupo de mayor prioridad, tendrá la misma prioridad que el operador punto (.), con asociatividad de izquierda a derecha. Además, este operador, como el operador punto, tendrá prioridad sobre los operadores unarios, aritméticos, relacionales, lógicos o de asignación que pueden aparecer en una expresión. Ya hemos discutido esto, cómo se aplica el operador punto, en la sección 11.2. Sin embargo, se deben hacer ciertas consideraciones con algunos operadores unarios, tales como ++, cuando se aplican a variables puntero a estructura.

Ya sabemos que expresiones como `++ptvar->miembro` y `++ptvar->miembro.submiembro` son equivalentes a `++(ptvar->miembro)` y `++(ptvar->miembro.submiembro)`, respectivamente. Así, tales expresiones producirán un incremento del valor del miembro o del submiembro, como se discutió en la sección 11.2. Por otra parte, la expresión `++ptvar` producirá el incremento del valor de `ptvar` en el número de bytes asociado con la estructura a la cual apunta `ptvar`. (El número de bytes asociado con una estructura particular puede determinarse mediante el uso del operador `sizeof`, como se ilustró en el Ejemplo 11.15.) Por tanto, la dirección representada por `ptvar` cambiará como resultado de esta expresión. Análogamente, la expresión `(++ptvar).miembro` hará que el valor de `ptvar` sea incrementado en ese número antes de acceder a `miembro`. Hay algún peligro en realizar operaciones como ésta porque `ptvar` es posible que ya no apunte a una variable de estructura, ya que su valor ha sido modificado.

EJEMPLO 11.23. A continuación se muestra una variación del sencillo programa en C mostrado en el Ejemplo 11.15.

```
#include <stdio.h>

main()
{
  typedef struct {
     int mes;
     int dia;
     int anio;
  } fecha;

  struct {
     int num_cuenta;
     char tipo_cuenta;
     char nombre[80];
     float saldo;
     fecha ultimopago;
  } cliente, *pt = &cliente;

  printf("Número de bytes (dec): %d\n", sizeof *pt);
  printf("Número de bytes (hex): %x\n\n", sizeof *pt);
  printf("Dirección de comienzo (hex): %x\n", pt);
  printf("Dirección incrementada (hex): %x", ++pt);
}
```

Observe que `pt` es una variable puntero cuyo objeto es la variable de estructura `cliente`.

La primera instrucción `printf` muestra el número de bytes asociados a `cliente` representado como una cantidad decimal. La segunda instrucción `printf` muestra este mismo valor como una cantidad en

hexadecimal. La tercera instrucción `printf` muestra el valor de `pt` (dirección de comienzo de `cliente`) en hexadecimal, mientras que la cuarta instrucción `printf` muestra lo que ocurre cuando se incrementa `pt`.

La ejecución del programa produce la siguiente salida:

```
Número de bytes (dec): 93
Número de bytes (hex): 5d

Dirección de comienzo (hex): f72
Dirección incrementada (hex): fcf
```

Así vemos que `cliente` necesita `93` bytes en decimal, que es `5d` en hexadecimal. El valor inicial asignado a `pt` (la dirección de comienzo de `cliente`) es `f72` en hexadecimal. Cuando se incrementa `pt`, su valor aumenta en `5d` bytes en hexadecimal, hasta `fcf`.

Es interesante modificar este programa reemplazando el array de caracteres `nombre[80]` por el puntero a carácter `*nombre` y ejecutar el programa. ¿Qué piensa usted que pasará?

11.5. PASO DE ESTRUCTURAS A UNA FUNCIÓN

Hay varias maneras de pasar información de una estructura a o desde una función. Se pueden transferir los miembros individuales, o las estructuras completas. El mecanismo para realizar las transferencias varía, dependiendo del tipo de transferencia (miembros individuales o estructuras completas) y la versión particular de C.

Los miembros individuales de una estructura se pueden pasar a una función como argumentos en la llamada a la función, y un miembro de una estructura puede ser devuelto mediante la instrucción `return`. Para hacer esto, cada miembro de la estructura se trata como una variable ordinaria unievaluada.

EJEMPLO 11.24. A continuación se muestra el esquema de la estructura de un programa en C. Este esquema hace uso de las declaraciones de estructuras presentadas anteriormente.

```
float ajustar(char nombre[], int num_cuenta, float saldo);
                                 /* prototipo de función */
main()
{
  typedef struct {               /* declaración de estructura */
     int mes;
     int dia;
     int anio;
  } fecha;

struct {                         /* declaración de estructura */
     int num_cuenta;
     char tipo_cuenta;
     char nombre[80];
     float saldo;
     fecha ultimopago;
  } cliente;
```

```
    . . . . .

    cliente.saldo = ajustar(cliente.nombre, cliente.num_cuenta,
                            cliente.saldo);
    . . . . .
}

float ajustar(char nombre[], int num_cuenta, float saldo)

{
    float nuevosaldo;          /* declaración de variable local */

    . . . . .

    nuevosaldo = . . . . . .;  /* ajustar el valor de saldo */

    . . . . .

    return(nuevosaldo);
}
```

Este esquema de programa ilustra la manera de cómo pueden pasarse los miembros de una estructura a una función. En particular, `cliente.nombre`, `cliente.num_cuenta` y `cliente_saldo` son pasados a la función `ajustar`. Dentro de `ajustar`, el valor asignado a `nuevosaldo` presumiblemente hace uso de la información pasada a la función. Entonces este valor es devuelto a `main`, donde es asignado al miembro de la estructura `cliente.saldo`.

Observe la declaración de la función dentro de `main`. Esta declaración se podría haber escrito sin los nombres de los argumentos, como sigue:

```
float ajustar(char [], int, float);
```

Algunos programadores prefieren esta forma, ya que evita la especificación de nombres de argumentos ficticios para datos que en realidad son miembros de estructura. Sin embargo, seguiremos utilizando prototipos completos de función para tener ventaja en la comprobación de errores.

Una estructura completa se puede transferir a una función pasando un puntero a la estructura como un argumento. En principio, esto es similar al procedimiento usado para transferir un array a una función. Sin embargo, debemos usar una notación de punteros explícita para representar que se pasa una estructura como un argumento.

Debe quedar claro que una estructura pasada de esta manera será pasada por *referencia* en vez de por *valor*. Por tanto, si cualquiera de los miembros de la estructura es modificado dentro de la función, las modificaciones serán reconocidas fuera de la función. De nuevo vemos una analogía directa con la transferencia de arrays a funciones.

EJEMPLO 11.25. Consideremos el programa sencillo en C mostrado a continuación.

```
#include <stdio.h>

typedef struct {
     char *nombre;
     int  num_cuenta;
```

```
      char  tipo_cuenta;
      float  saldo;
   } registro;

   void ajustar(registro *pt);      /* prototipo de función */

   main()     /* transferir un puntero a estructura a una función */
   {
      static registro cliente = {"Lázaro", 3333, 'A', 33.33};

      printf("%s %d %c %.2f\n", cliente.nombre, cliente.num_cuenta,
                                cliente.tipo_cuenta, cliente.saldo);
      ajustar(&cliente);
      printf("%s %d %c %.2f\n", cliente.nombre, cliente.num_cuenta,
                                cliente.tipo_cuenta, cliente.saldo);
   }

   void ajustar(registro *pt)       /* definición de función */

   {
      pt->nombre = "José";
      pt->num_cuenta = 9999;
      pt->tipo_cuenta = 'R';
      pt->saldo = 99.99;
      return;
   }
```

Este programa ilustra la transferencia de una estructura a una función pasando la dirección de la estructura (un puntero) a la función. En particular, `cliente` es una estructura del tipo `registro`, cuyos miembros tienen asignados un conjunto de valores. Estos valores iniciales se muestran cuando el programa empieza a ejecutarse. A continuación se pasa la dirección de la estructura a la función `ajustar`, donde se le asignan valores diferentes a los miembros de la estructura.

Dentro de `ajustar`, observe la declaración de argumento formal que define a `pt` como un puntero a una estructura del tipo `registro`. Observe también la instrucción vacía `return`; no se devuelve nada explícitamente desde `ajustar` hasta `main`.

Dentro de `main` vemos que los valores actuales de los miembros de `cliente` son mostrados de nuevo después de acceder a `ajustar`. Así el programa ilustra si los cambios realizados en `ajustar` repercuten o no en la parte del programa que hizo la llamada.

La ejecución del programa produce la siguiente salida:

```
Lázaro 3333 A 33.33
José 9999 R 99.99
```

Observe que los valores asignados a los miembros de `cliente` dentro de `ajustar` son reconocidos dentro de `main`, como se esperaba.

Un puntero a una estructura puede ser devuelto desde una función a la parte del programa que hizo la llamada. Esta característica puede ser útil cuando se pasen varias estructuras a una función y sólo una de ellas es devuelta.

EJEMPLO 11.26. Localización de registros de clientes. Aquí tenemos un programa que ilustra cómo se pasa un array de estructuras a una función y cómo se devuelve un puntero a una estructura particular.

Supóngase que especificamos un número de cuenta para un cliente particular, y a continuación localizamos y mostramos el registro completo para ese cliente. Cada registro de cliente estará mantenido en una estructura como en el último ejemplo. Sin embargo, ahora el conjunto completo de registros estará almacenado en un array llamado `cliente`. Observe que cada elemento del array será una estructura independiente.

La estrategia básica será introducir un número de cuenta y transferirlo con el array de registros a la función llamada `buscar`. Dentro de `buscar`, el número de cuenta será comparado con el número de cuenta almacenado dentro de cada registro, hasta que se produzca una coincidencia o hasta que se haya buscado en toda la lista de registros. Si se produce una coincidencia, se devuelve a `main` un puntero a ese elemento del array (la estructura que contiene el registro del cliente deseado) y son mostrados los contenidos del registro.

Si no se produce una coincidencia después de la búsqueda en todo el array, entonces la función devuelve el valor `NULL` (cero) a `main`. El programa muestra entonces un mensaje de error pidiendo al usuario que reintroduzca el número de cuenta. Esta búsqueda continuará hasta que se introduzca el valor cero como número de cuenta.

A continuación se muestra el programa completo. Dentro de este programa, `cliente` es un array de estructuras del tipo `registro` y `pt` un puntero a estructuras de este mismo tipo. La función `buscar` acepta dos argumentos y devuelve un puntero a una estructura del tipo `registro`. Los argumentos son un array de estructuras del tipo `registro` y una cantidad entera respectivamente. Dentro de `buscar`, la cantidad devuelta es o bien la dirección de un elemento del array o bien `NULL` (cero).

```
/* encontrar un registro de cliente que corresponda a un número de
   cuenta especificado */

#include <stdio.h>

#define N 3
#define NULL 0
typedef struct {
      char  *nombre;
      int  num_cuenta;
      char  tipo_cuenta;
      float  saldo;
} registro;

registro *buscar(registro tabla[], int ncuenta);  /* prototipo  de
                                                     función */
main()
{
   static registro cliente[N] = {
                                 {"Lázaro", 3333, 'A', 33.33},
                                 {"José", 6666, 'R', 66.66},
                                 {"Rafael", 9999, 'D', 99.99}
                              };    /* array de estructuras */
```

```
    int  ncuenta;                   /* declaración de variable */
    registro  *pt;                  /* declaración de puntero */

    printf("Localizador de cuenta de cliente\n");
    printf("Para FIN, introducir 0 para el número de cuenta\n");
    printf("\nCuenta nº: ");    /* introducir primer número de cuenta */
    scanf("%d", &ncuenta);

    while (ncuenta != 0) {
        pt = buscar(cliente, ncuenta);

        if (pt != NULL) {       /* se encontró una coincidencia */
            printf("\nNombre: %s\n", pt->nombre);
            printf("Cuenta nº: %d\n", pt->num_cuenta);
            printf("Tipo de cuenta: %c\n", pt->tipo_cuenta);
            printf("Saldo: %.2f\n", pt->saldo);
        }
        else
            printf("\nERROR - Por favor pruebe de nuevo\n");

        printf("\nCuenta nº: ");    /* introducir siguiente
                                        número de cuenta */
        scanf("%d", &ncuenta);
    }
}

registro *buscar(registro tabla[N], int ncuenta)    /* definición
                                                        de función */

/* acepta un array de estructuras y un número de cuenta, devuelve
   un puntero a una estructura particular (un elemento del array)
   si el número de cuenta coincide con un elemento del array */
{
   int cont;

   for (cont = 0; cont < N; ++cont)
        if (tabla[cont].num_cuenta == ncuenta)    /* encontrada
                                                      una coinci-
                                                      dencia */
            return(&tabla[cont]); /* devuelve un puntero al ele-
                                      mento del array */

   return(NULL);
}
```

El tamaño del array se expresa en términos de la constante simbólica `N`. Para este ejemplo hemos escogido el valor de `N = 3`. Esto es, estamos almacenando sólo tres registros de muestra en el array. En un ejemplo más real, `N` debería tener un valor mucho más grande.

Finalmente debe mencionarse que hay muchas maneras mejores de buscar en un conjunto de registros que ir examinando cada registro de modo secuencial. Se ha elegido este procedimiento simple, aunque ineficiente, para concentrarnos en el mecanismo de transferencia de estructuras entre `main` y la función subordinada `buscar`.

A continuación se muestra un diálogo típico que puede producirse por la ejecución del programa. Las respuestas del usuario están subrayadas, como siempre.

```
Localizador de cuenta de cliente
Para FIN, introducir 0 para el número de cuenta

Cuenta nº: 3333

Nombre: Lázaro
Cuenta nº: 3333
Tipo de cuenta: A
Saldo: 33.33

Cuenta nº: 9999

Nombre: Rafael
Cuenta nº: 9999
Tipo de cuenta: D
Saldo: 99.99

Cuenta nº: 666

ERROR - Por favor pruebe de nuevo

Cuenta nº: 6666

Nombre: José
Cuenta nº: 6666
Tipo de cuenta: R
Saldo: 66.66

Cuenta nº: 0
```

La mayoría de las nuevas versiones de C permiten que una estructura completa sea transferida directamente a una función como un argumento y devuelta directamente mediante la instrucción `return`. (Observe el contraste con los arrays, que *no pueden* ser devueltos mediante la instrucción `return`.) Estas características están incluidas en el nuevo estándar ANSI.

Cuando se pasa una estructura directamente a una función, la transferencia es por valor y no por referencia. Esto es consistente con las otras transferencias directas (sin punteros) en C. Por tanto, si cualquiera de los miembros de la estructura es modificado dentro de la función, las modificaciones *no serán* reconocidas fuera de la función. Sin embargo, si la estructura modificada es devuelta a la parte llamadora del programa, entonces los cambios *serán* reconocidos dentro de este ámbito mayor.

EJEMPLO 11.27. En el Ejemplo 11.25 vimos un programa que transfería un puntero a estructura a una función. Dos instrucciones `printf` distintas ilustraban que la transferencia de este tipo era por referencia, esto es, las modificaciones hechas a la estructura dentro de la función son reconocidas dentro de `main`. Un programa similar se muestra a continuación. Sin embargo, el programa actual transfiere una estructura completa a la función en vez de un puntero a estructura.

```
#include <stdio.h>

typedef struct {
     char *nombre;
     int num_cuenta;
     char tipo_cuenta;
     float saldo;
} registro;

void ajustar(registro cliente);          /* prototipo de función */

main()          /* transferir una estructura a una función */
{
   static registro cliente = {"Lázaro", 3333, 'A', 33.33};

   printf("%s %d %c %.2f\n", cliente.nombre, cliente.num_cuenta,
                             cliente.tipo_cuenta, cliente.saldo);
   ajustar(cliente);
   printf("%s %d %c %.2f\n", cliente.nombre, cliente.num_cuenta,
                             cliente.tipo_cuenta, cliente.saldo);
}

void ajustar(registro clien)   /* definición de función */

{
   clien.nombre = "José";
   clien.num_cuenta = 9999;
   clien.tipo_cuenta = 'R';
   clien.saldo = 99.99;
   return;
}
```

Observe que la función `ajustar` ahora acepta una estructura del tipo `registro` en vez de un puntero a ese tipo de estructura, como en el Ejemplo 11.25. En ninguno de los programas se devuelve nada desde `ajustar` a `main`.

Cuando se ejecuta el programa, se obtiene la siguiente salida:

```
Lázaro 3333 A 33.33
Lázaro 3333 A 33.33
```

Por tanto, las nuevas asignaciones hechas dentro de `ajustar` no son reconocidas dentro de `main`. Esto es lo esperado, ya que la transferencia de la estructura `cliente` a `ajustar` es por valor y no por referencia. (Comparar con la salida mostrada en el Ejemplo 11.25.)

Supongamos ahora que modificamos el programa de modo que la estructura modificada sea devuelta desde `ajustar` a `main`. Aquí tenemos el programa modificado.

```
#include <stdio.h>

typedef struct {
     char *nombre;
     int num_cuenta;
```

```
        char  tipo_cuenta;
        float  saldo;
  } registro;

  registro ajustar(registro cliente);     /* prototipo de función */

  main()          /* transferir una estructura a una función y devolver
                     la estructura */
  {
    static registro cliente = {"Lázaro", 3333, 'A', 33.33};

    printf("%s %d %c %.2f\n", cliente.nombre, cliente.num_cuenta,
                              cliente.tipo_cuenta, cliente.saldo);
    cliente = ajustar(cliente);
    printf("%s %d %c %.2f\n", cliente.nombre, cliente.num_cuenta,
                              cliente.tipo_cuenta, cliente.saldo);
  }

  registro ajustar(registro clien)  /* definición de función */

  {
    clien.nombre = "José";
    clien.num_cuenta = 9999;
    clien.tipo_cuenta = 'R';
    clien.saldo = 99.99;
    return(clien);
  }
```

Observe que `ajustar` devuelve ahora una estructura del tipo `registro` a `main`. La instrucción `return` se modifica adecuadamente.

La ejecución de este programa produce la siguiente salida:

```
Lázaro 3333 A 33.33
José 9999 R 99.99
```

Así, las modificaciones hechas dentro de `ajustar` son reconocidas dentro de `main`. Esto era lo esperado, ya que la estructura modificada se devuelve a la parte llamadora del programa. (Comparar con la salida mostrada en el Ejemplo 11.25 así como la mostrada anteriormente en este ejemplo.)

La mayoría de las versiones de C permiten que estructuras de datos complejas sean transferidas libremente entre funciones. Se han visto ejemplos que involucran la transferencia de miembros individuales de una estructura, estructuras completas, punteros a estructuras y arrays de estructuras. Sin embargo, debido a un problema práctico, hay algunas limitaciones sobre la complejidad de los datos que pueden ser fácilmente transferidos a o desde una función. En particular, algunos compiladores pueden tener dificultades al ejecutar programas que involucran transferencias de estructuras de datos complejas, debido a ciertas restricciones de memoria. El programador no experimentado debe saber que existen estas limitaciones, pero los detalles de este tema están fuera del ámbito de este texto.

EJEMPLO 11.28. Actualización de registros de clientes. El Ejemplo 11.14 presentó un sistema sencillo de facturación de clientes ilustrando la utilización de estructuras para mantener y actualizar registros de clientes. En dicho ejemplo los registros de clientes se almacenaban en un array global (externo) de estructuras. Consideremos ahora dos variaciones de este programa. En cada nuevo programa el array de estructuras es mantenido localmente, dentro de `main`. Los elementos individuales del array (registros individuales de clientes) son transferidos a o desde funciones según se requiera.

En el primer programa las estructuras completas son transferidas entre las funciones. En particular, la función `leerentrada` permite introducir en la computadora la información que define a cada registro de cliente. Cuando un registro entero ha sido introducido, su correspondiente estructura es devuelta a `main`, donde es almacenada en el array de `100` elementos llamado `cliente` y ajustado su tipo de cuenta. Después de que todos los registros hayan sido introducidos y ajustados, se transfieren de modo individual a la función `escribirsalida`, donde se muestra cierta información de cada cliente.

A continuación se muestra el programa completo.

```
/* actualizar una serie de cuentas de clientes (sistema de factu-
   ración simplificado) */

#include <stdio.h>

/* mantener las cuentas de clientes como un array de estructuras,
   transferir estructuras completas a y desde las funciones */

typedef struct {
     int mes;
     int dia;
     int anio;
} fecha;

typedef struct {
     char nombre[80];
     char calle[80];
     char ciudad[80];
     int num_cuenta;        /* (entero positivo) */
     int tipo_cuenta;       /* A (Al día), R (atrasada) o D (delin-
                               cuente) */
     float anteriorsaldo; /* (cantidad no negativa) */
     float nuevosaldo;    /* (cantidad no negativa) */
     float pago;          /* (cantidad no negativa) */
     fecha ultimopago;
} registro;

registro leerentrada(int i);               /* prototipo de función */
void escribirsalida(registro cliente); /* prototipo de función */

main()

/* leer cuentas de clientes, procesar cada cuenta y presentar la
   salida */
```

```
{
  int i, n;                /* declaración de variables */
  registro cliente[100]; /* declaración de array (array de es-
                             tructuras) */

  printf("SISTEMA DE FACTURACION DE CLIENTES\n\n");
  printf("¿Cuantos clientes hay? ");
  scanf("%d", &n);

  for (i = 0; i < n; ++i) {
      cliente[i] = leerentrada(i);

      /* determinar el estado de la cuenta */

      if (cliente[i].pago > 0)
         cliente[i].tipo_cuenta =
            (cliente[i].pago < 0.1 * cliente[i].anteriorsaldo) ? 'R'
                                              : 'A';
      else
         cliente[i].tipo_cuenta =
            (cliente[i].anteriorsaldo > 0) ? 'D' : 'A';

      /* ajustar el saldo de la cuenta */

       cliente[i].nuevosaldo   = cliente[i].anteriorsaldo
                               -  cliente[i].pago;
  }

  for (i = 0; i < n; ++i)
      escribirsalida(cliente[i]);
}

registro leerentrada(int i)

/* leer datos de entrada para un cliente */

{
  registro cliente;

  printf("\nCliente nº %d\n", i + 1);
  printf("    Nombre: ");
  scanf(" %[^\n]", cliente.nombre);
  printf("    Calle: ");
  scanf(" %[^\n]", cliente.calle);
  printf("    Ciudad: ");
  scanf(" %[^\n]", cliente.ciudad);
  printf("    Número de cuenta: ");
  scanf("%d", &cliente.num_cuenta);
  printf("    Saldo anterior: ");
  scanf("%f", &cliente.anteriorsaldo);
```

```
    printf("    Pago actual: ");
    scanf("%f", &cliente.pago);
    printf("    Fecha de pago (mm/dd/aaaa): ");
    scanf("%d/%d/%d", &cliente.ultimopago.mes,
                      &cliente.ultimopago.dia,
                      &cliente.ultimopago.anio);
    return(cliente);
}

void escribirsalida(registro cliente)

/* escribir la información actual para un cliente */

{
    printf("\nNombre:      %s", cliente.nombre);
    printf("       Número de cuenta: %d\n", cliente.num_cuenta);
    printf("Calle:   %s\n", cliente.calle);
    printf("Ciudad: %s\n\n", cliente.ciudad);
    printf("Saldo anterior: %7.2f", cliente.anteriorsaldo);
    printf("    Pago actual: %7.2f", cliente.pago);
    printf("    Nuevo saldo: %7.2f\n\n", cliente.nuevosaldo);
    printf("Estado de la cuenta: ");

    switch (cliente.tipo_cuenta) {
       case 'A':
                 printf("AL DIA\n\n");
                 break;
       case 'R':
                 printf("ATRASADA\n\n");
                 break;
       case 'D':
                 printf("DELINCUENTE\n\n");
                 break;
    }
    return;
}
```

El siguiente programa es similar al anterior. Sin embargo, ahora la transferencia involucra punteros a estructuras en vez de estructuras. Así las estructuras se transfieren ahora por referencia, mientras que en el programa anterior lo eran por valor.

Por brevedad, sólo se muestra el esquema del programa en vez del listado completo. Los bloques que faltan son idénticos a las partes correspondientes del programa anterior.

```
/* actualizar una serie de cuentas de clientes (sistema de factu-
   ración simplificado) */

#include <stdio.h>

/* mantener las cuentas de clientes como un array de estructuras,
   transferir punteros a estructuras a y desde las funciones */
```

```
/* (definiciones de estructuras) */

registro *leerentrada(int i);          /* prototipo de función */
void escribirsalida(registro *cliente); /* prototipo de función */

main()

/* leer cuentas de clientes, procesar cada cuenta y presentar la
   salida */

{
  int i, n;                   /* declaración de variables */
  registro cliente[100];      /* declaración de array (array de es-
                                 tructuras) */
  . . . . .

  for (i = 0; i < n; ++i) {
      cliente[i] = *leerentrada(i);

      /* determinar el estado de la cuenta */
      . . . . .
      /* ajustar el saldo de la cuenta */
      . . . . .
  }

  for (i = 0; i < n; ++i)
      escribirsalida(&cliente[i]);
}

registro *leerentrada(int i)

/* leer datos de entrada para un cliente */

{
  registro cliente;

  /* introducir datos */

  return(&cliente);

}

void escribirsalida(registro *pt)

/* escribir la información actual para un cliente */

{
  registro cliente;

  cliente = *pt;

  /* mostrar los datos de salida */

  return;
}
```

Ambos programas se comportan de la misma manera que el programa dado en el Ejemplo 11.14 cuando se ejecutan. Sin embargo, debido a la complejidad de las estructuras de datos (el array de estructuras, donde cada estructura contiene a su vez arrays y estructuras), los programas compilados pueden no ser ejecutables con ciertos compiladores. En particular, una condición de desbordamiento de pila (un tipo de condición de memoria inadecuada) puede ser experimentada con ciertos compiladores.

Este problema no existiría si el programa fuera más real, esto es, si los registros de clientes estuvieran almacenados dentro de un archivo en un dispositivo de memoria auxiliar, en vez de en un array que se almacena en la memoria de la computadora. Discutiremos este problema en el Capítulo 12, donde consideraremos el uso de archivos de datos para una situación como ésta.

11.6. ESTRUCTURAS AUTORREFERENCIADORAS

A veces es deseable incluir dentro de una estructura un miembro que sea un puntero a este tipo de estructura. En términos generales esto puede ser expresado como

```
struct marca {
    miembro 1;
    miembro 2;
    . . . . .
    struct marca *nombre;
};
```

donde *nombre* refiere el nombre de la variable puntero. Así, la estructura del tipo *marca* contendrá un miembro que apunta a otra estructura del tipo *marca*. Tales estructuras son conocidas como estructuras *autorreferenciadoras*.

EJEMPLO 11.29. Un programa en C contiene la siguiente declaración de estructura:

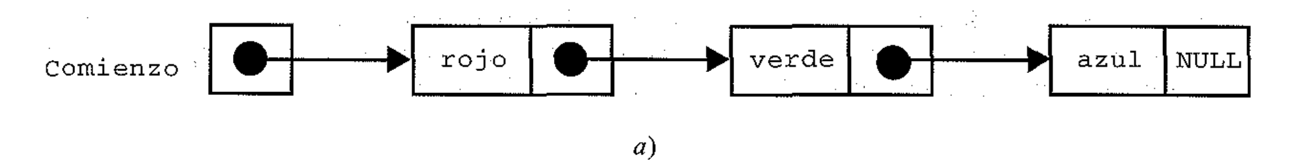
```
struct lista_elementos {
             char elem[40];
             struct lista_elementos *sig;
};
```

Ésta es una estructura del tipo `lista_elementos`. La estructura contiene dos miembros: un array de `40` caracteres, llamado `elem`, y un puntero a otra estructura del mismo tipo (un puntero a otra estructura del tipo `lista_elementos`) llamado `sig`. Por tanto, se trata de una estructura autorreferenciadora.

Las estructuras autorreferenciadoras son muy útiles en aplicaciones que involucren estructuras de datos enlazadas, tales como listas y árboles. Veremos un ejemplo completo que ilustra el procesamiento de una lista enlazada en el Ejemplo 11.32. Sin embargo, presentamos primero un breve resumen de estructuras de datos enlazadas.

La idea básica de una estructura de datos enlazada es que cada componente dentro de la estructura incluye un puntero indicando dónde está el siguiente componente. Por tanto, el orden relativo de los componentes puede ser fácilmente cambiado, simplemente modificando los punteros. Además, los componentes individuales pueden ser fácilmente añadidos o borrados, simplemente modificando los punteros. Como resultado, una estructura de datos enlazada no se confina a un número máximo de componentes. De esta forma, la estructura de datos puede expandir o contraer su tamaño según se necesite.

EJEMPLO 11.30. La Figura 11.3(*a*) muestra una lista enlazada con tres componentes. Cada componente consta de dos elementos: una cadena de caracteres y un puntero que referencia el siguiente componente dentro de la lista. Así, el primer componente contiene la cadena `rojo`, el segundo contiene `verde` y el tercero `azul`. El principio de la lista es indicado por un puntero separado, etiquetado como `comienzo`. También el final de la lista está marcado por un puntero especial llamado `NULL`.



a)

Añadamos ahora otro componente, cuyo valor es `blanco`, entre `rojo` y `verde`. Para hacer esto simplemente cambiamos los punteros, como se ilustra en le Figura 11.3(*b*). Análogamente, si elegimos borrar el elemento cuyo valor es `verde`, sencillamente cambiamos el puntero asociado con el segundo componente, como se muestra en la Figura 11.3(*c*).

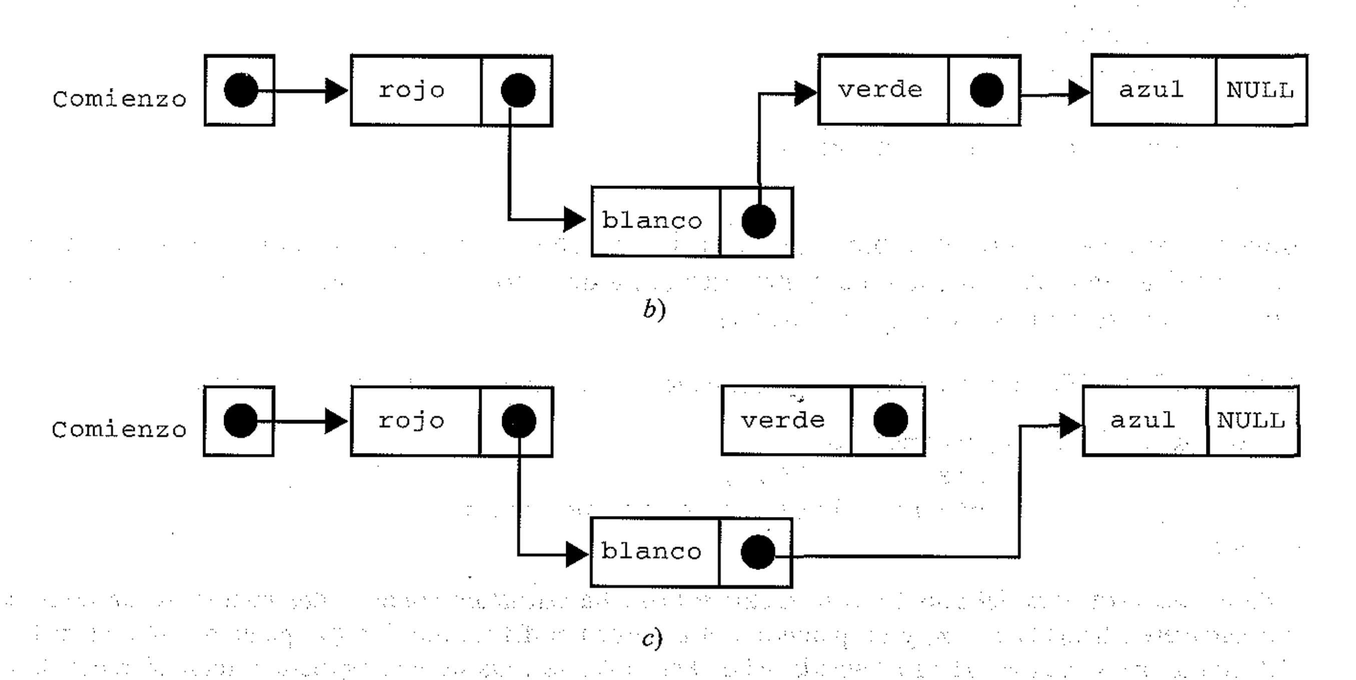


Figura 11.3.

Hay varios tipos distintos de estructuras de datos enlazadas, incluyendo listas *lineales*, en las que los componentes están todos enlazados de un modo secuencial; listas enlazadas con *múltiples punteros*, que permiten recorrer la lista hacia delante o hacia atrás; listas enlazadas *circulares*, que no tienen ni principio ni fin, y *árboles*, en los cuales los componentes se ordenan jerárquicamente. Ya hemos visto una ilustración de una lista lineal enlazada en el Ejemplo 11.30. Otros tipos de listas enlazadas se ilustran en el siguiente ejemplo.

EJEMPLO 11.31. En la Figura 11.4 vemos una lista lineal enlazada similar a la mostrada en la Figura 11.3(*a*). Sin embargo, ahora hay *dos* punteros asociados con cada componente: un puntero al si-

guiente y un puntero al anterior. Este doble conjunto de punteros permite recorrer la lista en cualquier dirección, de principio a fin, o del fin al principio.

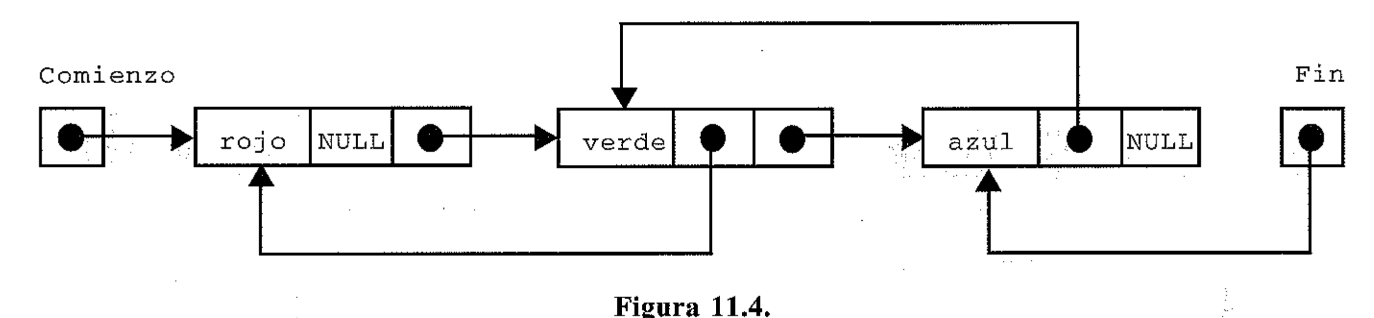


Figura 11.4.

Consideremos ahora la lista mostrada en la Figura 11.5. Esta lista es similar a la mostrada en la Figura 11.3(*a*), excepto que el último elemento (`azul`) apunta al primer elemento (`rojo`). Por tanto esta lista no tiene principio ni fin. Tales listas son denominadas *listas circulares*.

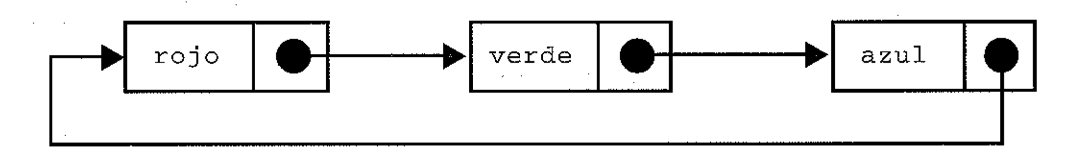


Figura 11.5.

Finalmente, en la Figura 11.6(*a*) se ve un ejemplo de un *árbol*. Los árboles se componen de nodos y ramas, dispuestos de algún modo jerárquico, que indica la correspondiente estructura jerárquica dentro de los datos. (Un *árbol binario* es un árbol en el cual cada nodo no tiene más de dos ramas.)

En la Figura 11.6(*a*) el *nodo raíz* tiene el valor `pantalla` y las ramas asociadas conducen a los nodos cuyos valores son `primerplano` y `fondo`, respectivamente. Análogamente, las ramas asociadas con `primerplano` conducen a los nodos cuyos valores son `blanco`, `verde` y `ámbar`, y las ramas asociadas con `fondo` conducen a los nodos cuyos valores son `negro`, `azul` y `blanco`.

La Figura 11.6(*b*) ilustra la manera de cómo se usan los punteros para construir el árbol.

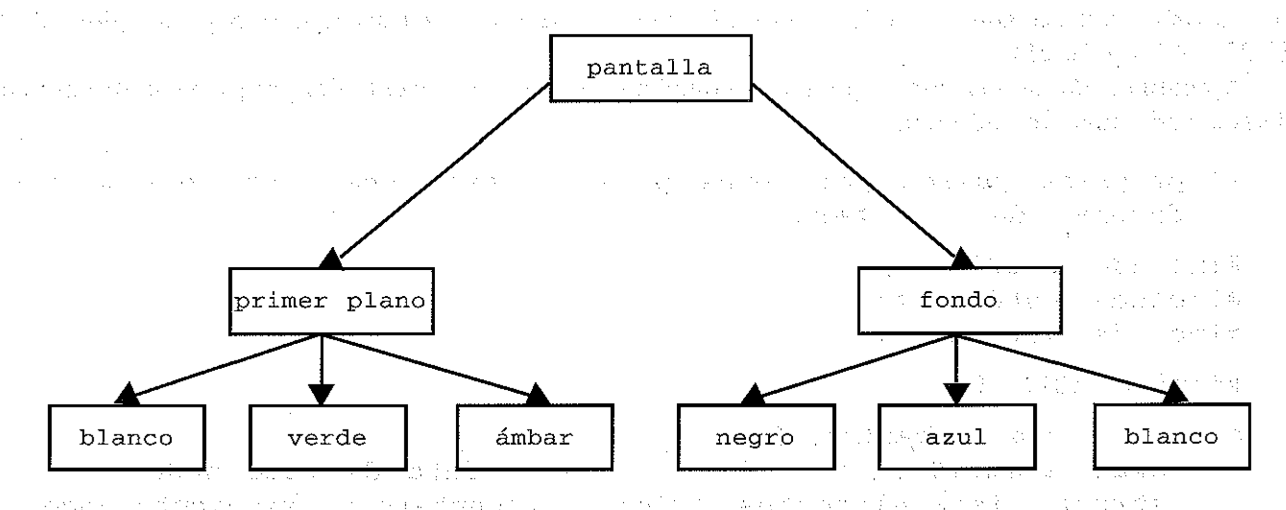


Figura 11.6. *a)*

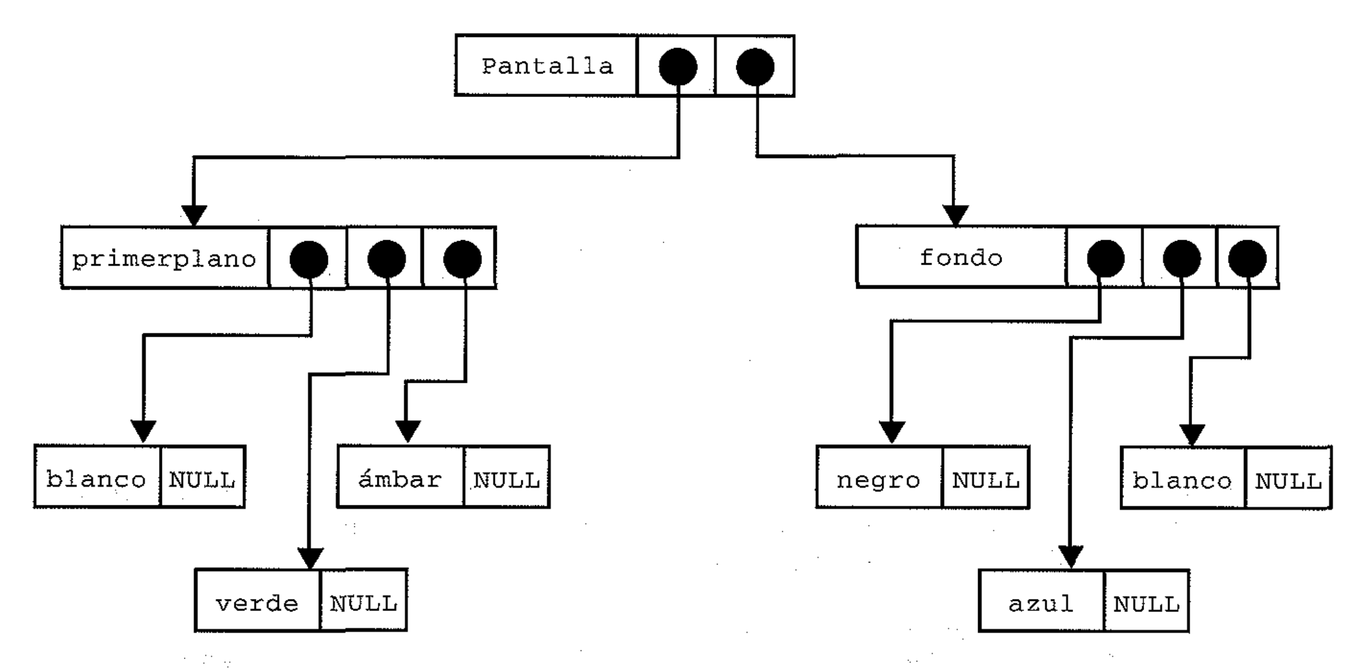


Figura 11.6. *b*)

Las estructuras autorreferenciadoras son idealmente convenientes para aplicaciones que involucren estructuras de datos enlazadas. Cada estructura representará un componente simple (un nodo) dentro de la estructura de datos enlazada. El puntero autorreferenciador indicará la localización del siguiente componente.

EJEMPLO 11.32. Procesamiento de una lista enlazada. Presentamos ahora un programa interactivo que permite crear una lista lineal enlazada, añadir nuevos elementos a la lista o borrar elementos de la lista enlazada. Cada componente consistirá en una cadena de caracteres y un puntero al siguiente componente. El programa funcionará dirigido por menús para facilitar su uso a los no programadores. Incluiremos la posibilidad de mostrar la lista después de la selección de cualquier elemento del menú (después de cualquier cambio hecho a la lista).

Este programa es algo más complejo que los programas de ejemplos precedentes. Se utiliza la *recursividad* (ver sección 7.6) y la *gestión dinámica de memoria* (ver sección 10.5 y Ejemplos 10.15, 10.22, 10.24 y 10.26).

A continuación se muestra el programa completo. Después del listado del programa se discute con detalle cada función individual.

```
/* programa guiado por menús para procesar una lista enlazada de
   cadenas de caracteres */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define NULL 0

struct lista_elementos {
    char elem[40];                  /* dato de este nodo */
    struct lista_elementos *sig;    /*puntero al siguiente nodo */
};
```

```
typedef struct lista_elementos nodo; /* declaración de tipo de es-
                                        tructura */

int menu(void);                         /* prototipo de función */
void crear(nodo *pt);                   /* prototipo de función */
nodo *insertar(nodo *pt);               /* prototipo de función */
nodo *eliminar(nodo *pt);               /* prototipo de función */
void mostrar(nodo *pt);                 /* prototipo de función */

main()
{
  nodo *prin;                  /* puntero al principio de la lista */
  int eleccion;                /* declaración de variable local */

  do {
        eleccion = menu();
        switch(eleccion) {

        case 1:   /* crear la lista enlazada */
              prin = (nodo *) malloc(sizeof(nodo)); /* reserva  de
                                                      espacio para
                                                      el    primer
                                                      nodo */
              crear(prin);
              printf("\n");
              mostrar(prin);
              continue;

        case 2:   /* añadir un componente */
              prin = insertar(prin);
              printf("\n");
              mostrar(prin);
              continue;

        case 3:   /* borrar un componente */
              prin = eliminar(prin);
              printf("\n");
              mostrar(prin);
              continue;

        default:    /* finalizar */
              printf("Fin de las operaciones\n");
        }
  } while (eleccion != 4);
}
```

```
int menu(void)    /* generar el menú principal */

{
  int eleccion;

  do {
        printf("\nMenú principal:\n");
        printf("   1 - CREAR la lista enlazada\n");
        printf("   2 - AÑADIR un componente\n");
        printf("   3 - BORRAR un componente\n");
        printf("   4 - FIN\n");
        printf("Por favor, introduzca su elección (1, 2, 3 o 4) -> ");
        scanf("%d", &eleccion);
        if (eleccion < 1 || eleccion > 4)
             printf("\nERROR - Por favor, inténtelo de nuevo\n");
  } while (eleccion < 1 || eleccion > 4);
  printf("\n");
  return(eleccion);
}

void crear(nodo *registro)   /* crear una lista enlazada */

/* el argumento apunta al nodo actual */

{
  printf("Dato (escribir \'FIN\' para terminar): ");
  scanf(" %[^\n]", registro->elem);

  if (strcmp(registro->elem, "FIN") == 0)
     registro->sig = NULL;
  else {
     /* reservar espacio para el siguiente nodo */
     registro->sig = (nodo *) malloc(sizeof(nodo));

     /* crear el siguiente nodo */
     crear(registro->sig);
  }
  return;
}

void mostrar(nodo *registro) /* mostrar la lista enlazada */

/* el argumento apunta al nodo actual */

{
  if (registro->sig != NULL) {
     printf("%s\n", registro->elem); /* mostrar este dato */
     mostrar(registro->sig);         /* tomar el siguiente elemento */
  }
  return;
}
```

```
nodo *insertar(nodo *primero) /* añade un componente a la lista en-
                                 lazada; devuelve un puntero al prin-
                                 cipio de la lista modificada */

/* el argumento apunta al primer nodo */

{
  nodo *localizar(nodo *, char[]); /* declaración de función */
  nodo *nuevoregistro;             /* puntero al nuevo nodo */
  nodo *marca;                     /* puntero al nodo ANTERIOR
                                      al buscado */
  char nuevodato[40];              /* dato nuevo */
  char objetivo[40];               /* dato siguiente a la nueva
                                      entrada */

  printf("Nuevo dato: ");
  scanf(" %[^\n]", nuevodato);
  printf("Colocar delante de (escribir \'FIN\' si es el último): ");
  scanf(" %[^\n]", objetivo);

  if (strcmp(primero->elem, objetivo) == 0) {
     /* el nuevo nodo es el primero de la lista */

     /* reservar espacio para el nuevo nodo */
     nuevoregistro = (nodo *) malloc(sizeof(nodo));

     /* asignar el nuevo dato a nuevoregistro->elem); */
     strcpy(nuevoregistro->elem, nuevodato);

     /* asignar el puntero actual a nuevoregistro->sig */
     nuevoregistro->sig = primero;

     /* el nuevo puntero se convierte en el principio de la lista */
     primero = nuevoregistro;
  }

  else {
     /* insertar el nuevo nodo tras algún nodo existente */

     /* localizar el nodo PRECEDENTE del nodo objetivo */
     marca = localizar(primero, objetivo);

     if (marca == NULL)
       printf("\nNo se encuentra coincidencia - Pruebe de nuevo\n");
     else {
        /* reservar espacio para el nuevo nodo */
        nuevoregistro = (nodo *) malloc(sizeof(nodo));

        /* asignar el nuevo dato a nuevoregistro->elem */
        strcpy(nuevoregistro->elem, nuevodato);
```

```
            /* asignar el siguiente puntero a nuevoregistro->sig */
            nuevoregistro->sig = marca->sig;

            /* asignar el nuevo puntero a marca->sig */
            marca->sig = nuevoregistro;
        }
    }
    return(primero);
}

nodo *localizar(nodo *registro, char objetivo[]) /* localizar un
                                                    nodo */

/*  devuelve un puntero al nodo ANTERIOR al nodo objetivo
    El primer argumento apunta al nodo actual
    El segundo argumento es la cadena objetivo  */
{
    if (strcmp(registro->sig->elem, objetivo) == 0)  /*coinci-
                                                      dencia en-
                                                      contrada */
        return(registro);
    else
       if (registro->sig->sig == NULL)            /* fin de lista */
           return(NULL);
       else
           localizar(registro->sig, objetivo);   /* probar el
                                                  siguiente nodo */
}

nodo *eliminar(nodo *primero)    /* eliminar (borrar) un componen-
                                    te de la lista enlazada; de-
                                    vuelve un puntero al princi-
                                    pio de la lista modificada */

/* el argumento apunta al primer nodo */

{
    nodo *localizar(nodo *, char[]);  /* declaración de función */
    nodo *marca;                      /* puntero al nodo ANTERIOR
                                         al buscado */
    nodo *temp;                       /* puntero temporal */
    char objetivo[40];                /* dato a borrar */

    printf("Dato a borrar: ");
    scanf(" %[^\n]", objetivo);

    if (strcmp(primero->elem, objetivo) == 0) {
       /* borrar el primer nodo */
```

```
        /* marcar el nodo siguiente al nodo objetivo */
        temp = primero->sig;

        /* liberar el espacio del nodo objetivo */
        free(primero);

        /* ajustar el puntero al primer nodo */
        primero = temp;
    }

    else {
        /* borrar otro dato distinto del primero */

        /* localizar el nodo PRECEDENTE del nodo objetivo */
        marca = localizar(primero, objetivo);

        if (marca == NULL)
           printf("\nNo se encuentra coincidencia - Pruebe de nuevo\n");

    else {
        /* marcar el nodo siguiente al nodo objetivo */
        temp = marca->sig->sig;

        /* liberar el espacio del nodo objetivo */
        free(marca->sig);

        /* ajustar el enlace para el siguiente nodo */
        marca->sig = temp;
    }
  }
  return(primero);
}
```

El programa comienza con las instrucciones `#include` usuales y una definición de la constante simbólica `NULL` para representar el valor `0`. Siguiendo a estas instrucciones está la declaración de la estructura autorreferenciadora `lista_elementos`. Esta declaración es la misma que la mostrada en el Ejemplo 11.29. Así, `lista_elementos` identifica una estructura con dos miembros: un array de `40` caracteres (`elem`) y un puntero (`sig`) a otra estructura del mismo tipo. El array de caracteres representará una cadena y el puntero indicará la localización de la siguiente componente de la lista enlazada.

Luego se define el tipo de datos `nodo`, el cual identifica a estructuras del tipo `lista_elementos`. Esta definición es seguida por los prototipos de las funciones. Dentro de los prototipos de las funciones, observe que `pt` es un puntero a una estructura del tipo `nodo`. Este puntero indicará el comienzo de la lista enlazada. El resto de los prototipos de funciones identifican varias funciones adicionales que son llamadas desde `main`. Observe que estas declaraciones y los prototipos de las funciones son externas. Por tanto, serán reconocidas en todo el programa.

La función `main` consta de un bucle `do - while` que permite la ejecución repetida del proceso completo. Este bucle llama a la función `menu`, que genera el menú principal y devuelve un valor para `eleccion`, indicando la selección del usuario. Una instrucción `switch` llama entonces a la función apropiada en relación con la selección del usuario. Observe que el programa dejará de ejecutarse si `eleccion` tiene asignado el valor `4`.

Si eleccion tiene asignado el valor 1, indicando que se creará una nueva lista enlazada, se debe reservar un bloque de memoria para el primer elemento antes de llamar a la función crear. Esto se realiza usando la función de biblioteca malloc, como se discute en la sección 10.5. Así, la instrucción de reserva de memoria

```
prin = (nodo *) malloc(sizeof(nodo));
```

reserva un bloque de memoria cuyo tamaño (en bytes) es suficiente para un nodo. La instrucción devuelve un puntero a una estructura del tipo nodo. Este puntero indica el principio de la lista enlazada. Así se pasa a crear como argumento.

Observe que la conversión de tipo (nodo *) es requerida como una parte de la instrucción de reserva de memoria dinámica. Sin la misma, la función malloc devolvería un puntero a char en vez de un puntero a una estructura del tipo nodo.

Consideremos ahora la función menu, que se utiliza para generar el menú principal. Esta función acepta un valor para eleccion después de que el menú sea generado. Los únicos valores permitidos para eleccion son 1, 2, 3 o 4. Una comprobación de errores, mediante una instrucción do - while, hace que se muestre un mensaje de error y que se genere un nuevo menú si se introduce otro valor distinto de 1, 2, 3 o 4 en respuesta al menú.

La lista enlazada se crea mediante la función crear. Ésta es una función recursiva que acepta un puntero al nodo actual (el nodo que ha sido creado) como argumento. La variable puntero se llama registro.

La función crear comienza pidiendo el dato actual; esto es, la cadena de caracteres que va a residir en el nodo actual. Si el usuario introduce la cadena FIN (en mayúsculas o en minúsculas), entonces se le asigna NULL al puntero que indica la localización del siguiente nodo y termina la recursividad. Sin embargo, si el usuario introduce otra cadena de caracteres distinta de FIN, se reserva la memoria para el siguiente nodo mediante la función malloc y la función se llama a sí misma de modo recursivo. Así la recursividad continuará hasta que el usuario introduzca FIN para uno de los elementos.

Una vez que la lista enlazada ha sido creada, se muestra mediante la función mostrar. Esta función se llama desde main, después de llamar a crear. Observe que mostrar acepta un puntero al nodo actual como argumento. La función se ejecuta recursivamente hasta que recibe un puntero cuyo valor es NULL. Por tanto, la recursividad hace que se muestre la lista enlazada completa.

Consideremos ahora la función insertar, que se usa para añadir una nueva componente (un nuevo nodo) a la lista enlazada. Esta función pregunta al usuario dónde se produce la inserción. Observe que la función acepta un puntero al principio de la lista como argumento y devuelve un puntero al principio de la lista después de haber realizado la inserción. Estos dos punteros serán los mismos salvo que la inserción sea realizada al principio de la lista.

La función insertar no se ejecuta recursivamente. Primero pide el nuevo dato (nuevodato), y seguidamente pide el dato existente que seguirá al nuevo dato (el dato existente se llama objetivo). Si la inserción se realiza al *principio de la lista*, se reserva memoria para el nuevo nodo, nuevodato es asignado al primer miembro y el puntero original que indicaba el comienzo de la lista enlazada (primero) es asignado al segundo miembro. El puntero devuelto por malloc, que indica el principio del nuevo nodo, es asignado a primero. Por tanto, el principio del nuevo nodo se convierte en el principio de toda la lista.

Si la inserción se realiza *detrás de un nodo existente*, entonces la función localizar se llama para determinar la localización de la inserción. Esta función devuelve un puntero al nodo que *precede* al nodo objetivo. El valor devuelto es asignado al puntero marca. Por tanto, marca apunta al nodo que precederá al nuevo nodo. Si localizar no encuentra una coincidencia entre el valor introducido para objetivo y un dato existente, entonces se devolverá NULL.

Si localizar encuentra una coincidencia, la inserción se hace de la siguiente manera: se reser-

va memoria para el nuevo nodo, `nuevodato` se asigna al primer miembro de `nuevoregistro` (a `nuevoregistro->elem`) y el puntero al nodo objetivo (`marca->sig`) se asigna al segundo miembro de `nuevoregistro` (`nuevoregistro->sig`). Entonces el puntero devuelto por `malloc`, que indica el principio del nuevo nodo, es asignado a `marca->sig`. Por tanto, el puntero en el nodo precedente apuntará al nuevo nodo y el puntero en el nuevo nodo apuntará al nodo objetivo.

Consideremos ahora la función `localizar`. Es una simple función recursiva que acepta un puntero al nodo actual y la cadena objetivo como argumentos y devuelve un puntero al nodo que *precede* al nodo actual. Por tanto, si el dato en el siguiente nodo al actual coincide con la cadena objetivo, la función devolverá el puntero al nodo actual. En otro caso, se puede tomar una de dos opciones posibles. Si el puntero en el siguiente nodo al nodo actual es `NULL`, que indica el final de la lista enlazada, no se ha producido una coincidencia. Por tanto, la función devolverá el valor `NULL`. Pero si el puntero en el siguiente nodo al actual es diferente de `NULL`, la función se llamará recursivamente, comprobando si se produce una coincidencia con el siguiente nodo.

Finalmente consideraremos la función `eliminar`, que se usa para borrar una componente existente (un nodo existente) de la lista enlazada. Esta función es similar a `insertar`, pero algo más simple. Acepta un puntero al principio de la lista enlazada como argumento y devuelve un puntero al principio de la lista enlazada después de borrar el elemento.

La función `eliminar` empieza pidiendo el dato a borrar (`objetivo`). Si éste es el primer dato, entonces los punteros son ajustados como sigue: el puntero que indica la localización del segundo nodo es temporalmente asignado a la variable puntero `temp`; la memoria utilizada por el primer nodo es liberada, usando la función de biblioteca `free`; y la localización del segundo nodo (que es ahora el primer nodo debido al borrado) se asigna a `primero`. Por tanto, el principio del segundo (anterior) nodo se convierte en el principio de la lista completa.

Si el dato a borrar *no* es el primero de la lista, entonces se llama a `localizar` para determinar la localización del elemento a borrar. Esta función devolverá un puntero al nodo *precedente* al nodo objetivo. El valor devuelto se asigna a la variable puntero `marca`. Si el valor es `NULL`, no se produce una coincidencia. Se genera un mensaje de error pidiendo al usuario que lo intente nuevamente.

Si `localizar` devuelve un valor distinto de `NULL`, el nodo objetivo se borra de la siguiente manera: el puntero al nodo siguiente al nodo objetivo se asigna temporalmente a la variable puntero `temp`; la memoria utilizada por el nodo objetivo se libera, usando la función de biblioteca `free`; y el valor de `temp` se asigna a `marca->sig`. Por tanto, el puntero en el nodo precedente apuntará al nodo siguiente al nodo objetivo.

Utilicemos este programa para crear una lista enlazada que contenga las siguientes ciudades: Boston, Chicago, Denver, New York, Pittsburgh y San Francisco. Entonces añadiremos y borraremos varias ciudades, ilustrando así todas las características del programa. Mantendremos la lista de ciudades ordenada alfabéticamente a lo largo del ejercicio. (Por supuesto, podríamos hacer que la computadora realizara la ordenación por nosotros, pero sería complicar más un programa que ya es complejo.)

A continuación se muestra la sesión interactiva. Como siempre las respuestas del usuario han sido subrayadas.

```
Menú principal:
   1 - CREAR la lista enlazada
   2 - AÑADIR un componente
   3 - BORRAR un componente
   4 - FIN
Por favor, introduzca su elección (1, 2, 3, 4) -> 1

Dato (escribir 'FIN' para terminar): BOSTON
Dato (escribir 'FIN' para terminar): CHICAGO
```

```
Dato (escribir 'FIN' para terminar): DENVER
Dato (escribir 'FIN' para terminar): NEW YORK
Dato (escribir 'FIN' para terminar): PITTSBURGH
Dato (escribir 'FIN' para terminar): SAN FRANCISCO
Dato (escribir 'FIN' para terminar): FIN

BOSTON
CHICAGO
DENVER
NEW YORK
PITTSBURGH
SAN FRANCISCO

Menú principal:
   1 - CREAR la lista enlazada
   2 - AÑADIR un componente
   3 - BORRAR un componente
   4 - FIN
Por favor, introduzca su elección (1, 2, 3, 4) -> 2

Nuevo dato: ATLANTA
Colocar delante de (escribir 'FIN' si es el último): BOSTON

ATLANTA
BOSTON
CHICAGO
DENVER
NEW YORK
PITTSBURGH
SAN FRANCISCO

Menú principal:
   1 - CREAR la lista enlazada
   2 - AÑADIR un componente
   3 - BORRAR un componente
   4 - FIN
Por favor, introduzca su elección (1, 2, 3, 4) -> 2

Nuevo dato: SEATTLE
Colocar delante de (escribir 'FIN' si es el último): FIN

ATLANTA
BOSTON
CHICAGO
DENVER
NEW YORK
PITTSBURGH
SAN FRANCISCO
SEATTLE
```

```
Menú principal:
   1 - CREAR la lista enlazada
   2 - AÑADIR un componente
   3 - BORRAR un componente
   4 - FIN
Por favor, introduzca su elección (1, 2, 3, 4) -> 3

Dato a borrar: NEW YORK

ATLANTA
BOSTON
CHICAGO
DENVER
PITTSBURGH
SAN FRANCISCO
SEATTLE

Menú principal:
   1 - CREAR la lista enlazada
   2 - AÑADIR un componente
   3 - BORRAR un componente
   4 - FIN
Por favor, introduzca su elección (1, 2, 3, 4) -> 2

Nuevo dato: WASHINGTON
Colocar delante de (escribir 'FIN' si es el último): WILLIAMSBURG

No se encuentra coincidencia - Pruebe de nuevo

ATLANTA
BOSTON
CHICAGO
DENVER
PITTSBURGH
SAN FRANCISCO
SEATTLE

Menú principal:
   1 - CREAR la lista enlazada
   2 - AÑADIR un componente
   3 - BORRAR un componente
   4 - FIN
Por favor, introduzca su elección (1, 2, 3, 4) -> 2

Nuevo dato: WASHINGTON
Colocar delante de (escribir 'FIN' si es el último): FIN

ATLANTA
BOSTON
CHICAGO
DENVER
```

```
PITTSBURGH
SAN FRANCISCO
SEATTLE
WASHINGTON

Menú principal:
   1 - CREAR la lista enlazada
   2 - AÑADIR un componente
   3 - BORRAR un componente
   4 - FIN
Por favor, introduzca su elección (1, 2, 3, 4) -> 3

Dato a borrar: ATLANTA

BOSTON
CHICAGO
DENVER
PITTSBURGH
SAN FRANCISCO
SEATTLE
WASHINGTON

Menú principal:
   1 - CREAR la lista enlazada
   2 - AÑADIR un componente
   3 - BORRAR un componente
   4 - FIN
Por favor, introduzca su elección (1, 2, 3, 4) -> 2

Nuevo dato: DALLAS
Colocar delante de (escribir 'FIN' si es el último): DENVER

BOSTON
CHICAGO
DALLAS
DENVER
PITTSBURGH
SAN FRANCISCO
SEATTLE
WASHINGTON

Menú principal:
   1 - CREAR la lista enlazada
   2 - AÑADIR un componente
   3 - BORRAR un componente
   4 - FIN
Por favor, introduzca su elección (1, 2, 3, 4) -> 3

Dato a borrar: MIAMI

No se encuentra coincidencia - Pruebe de nuevo
```

```
BOSTON
CHICAGO
DALLAS
DENVER
PITTSBURGH
SAN FRANCISCO
SEATTLE
WASHINGTON

Menú principal:
   1 - CREAR la lista enlazada
   2 - AÑADIR un componente
   3 - BORRAR un componente
   4 - FIN
Por favor, introduzca su elección (1, 2, 3, 4) -> 3

Dato a borrar: WASHINGTON

BOSTON
CHICAGO
DALLAS
DENVER
PITTSBURGH
SAN FRANCISCO
SEATTLE

Menú principal:
   1 - CREAR la lista enlazada
   2 - AÑADIR un componente
   3 - BORRAR un componente
   4 - FIN
Por favor, introduzca su elección (1, 2, 3, 4) -> 5

ERROR - Por favor, inténtelo de nuevo

Menú principal:
   1 - CREAR la lista enlazada
   2 - AÑADIR un componente
   3 - BORRAR un componente
   4 - FIN
Por favor, introduzca su elección (1, 2, 3, 4) -> 4

Fin de las operaciones
```

11.7. UNIONES

Las uniones, como las estructuras, contienen miembros cuyos tipos de datos pueden ser diferentes. Sin embargo, los miembros que componen una unión comparten el mismo área de almacenamiento dentro de la memoria de la computadora, mientras que cada miembro dentro de la estruc-

tura tiene asignada su propia área de almacenamiento. Así, las uniones se usan para ahorrar memoria. Son útiles para aplicaciones que involucren múltiples miembros donde no se necesita asignar valores a todos los miembros a la vez.

Dentro de la unión, la reserva de memoria requerida para almacenar miembros cuyos tipos de datos son diferentes (con diferentes requisitos de memoria) es manejada automáticamente por el compilador. Sin embargo, el usuario debe conservar una pista del tipo de información que está almacenada en cada momento. Una tentativa de acceso al tipo de información equivocada producirá resultados sin sentido.

En términos generales, la composición de una unión puede definirse como

```
union  marca  {
   miembro 1;
   miembro 2;
   . . . . .
   miembro m;
};
```

donde `union` es una palabra reservada requerida y los otros términos tienen el mismo significado que en una definición de estructura (ver sección 11.1). Las variables de unión individuales pueden ser declaradas como

```
tipo-almacenamiento  union  marca
                      variable 1, variable 2, . . ., variable n;
```

donde *`tipo-almacenamiento`* es un especificador opcional de tipo de almacenamiento, `union` es la palabra reservada requerida, *`marca`* el nombre que aparece en la definición de la unión, y *`variable 1, variable 2, . . ., variable n`* son variables de unión del tipo *`marca`*.

Las dos declaraciones pueden ser combinadas, como se hizo con las estructuras. Así, se puede escribir

```
tipo-almacenamiento  union  marca  {
                miembro 1;
                miembro 2;
                . . . . .
                miembro m;
} variable 1, variable 2, . . ., variable n;
```

La *`marca`* es opcional en este tipo de declaración.

EJEMPLO 11.33. Un programa en C contiene la siguiente declaración de unión:

```
union  id  {
   char  color[12];
   int  talla;
} camisa, blusa;
```

Aquí tenemos dos variables de unión, `camisa` y `blusa`, del tipo `id`. Cada variable puede representar bien una cadena de 12 caracteres (`color`) o bien un entero (`talla`) en un momento dado.

La cadena de 12 caracteres requerirá más área de almacenamiento dentro de la memoria de la computadora que el entero. Por tanto, se reserva un bloque de memoria suficiente para almacenar la cadena de 12 caracteres para cada variable de unión. El compilador distinguirá automáticamente entre el array de 12 caracteres y el entero según se requiera.

Una unión puede ser un miembro de una estructura y una estructura puede ser un miembro de una unión. Además, las estructuras y las uniones pueden ser mezcladas libremente con los arrays.

EJEMPLO 11.34. Un programa en C contiene las siguientes declaraciones:

```
union  id  {
    char color[12];
    int talla;
};

struct  ropa  {
    char fabricante[20];
    float coste;
    union id descripcion;
} camisa, blusa;
```

Ahora `camisa` y `blusa` son variables de estructura del tipo `ropa`. Cada variable contendrá los siguientes miembros: una cadena de caracteres (`fabricante`), una cantidad en coma flotante (`coste`) y una unión (`descripcion`). La unión puede representar o bien una cadena de caracteres (`color`) o bien una cantidad entera (`talla`).

Otra forma para declarar las variables de estructura `camisa` y `blusa` es combinar las dos declaraciones anteriores como sigue:

```
struct  ropa  {
    char fabricante[20];
    float coste;
    union  {
        char color[12];
        int talla;
    } descripcion;
} camisa, blusa;
```

Esta declaración es más concisa, pero quizá menos directa que las declaraciones originales.

Un miembro individual de una unión puede ser accedido de la misma manera que un miembro de una estructura, usando los operadores . y ->. Así, si *variable* es una variable de unión, entonces *variable.miembro* refiere al miembro de la unión. Análogamente, si *ptvar* es una variable puntero que apunta a una unión, entonces *ptvar->miembro* refiere al miembro de esa unión.

EJEMPLO 11.35. Consideremos el sencillo programa en C mostrado a continuación.

```
#include <stdio.h>

main()
{
  union id {
      char color;
      int talla;
  };

  struct {
      char fabricante[20];
      float coste;
      union id descripcion;
  } camisa, blusa;

  printf("%d\n", sizeof(union id));

  /* asignar un valor a color */
  camisa.descripcion.color = 'B';
  printf("%c %d\n", camisa.descripcion.color, camisa.descripcion.talla);

  /* asignar un valor a talla */
  camisa.descripcion.talla = 12;
  printf("%c %d\n", camisa.descripcion.color, camisa.descripcion.talla);
}
```

Este programa contiene unas declaraciones similares a las mostradas en el Ejemplo 11.34. Sin embargo, observe que el primer miembro de la unión es ahora un carácter en vez del array de 12 caracteres del ejemplo anterior. Este cambio se ha hecho para simplificar la asignación de valores apropiados a los miembros de la unión.

Siguiendo a las declaraciones y a la instrucción `printf` inicial, vemos que se le asigna el carácter `'B'` al miembro de la unión `camisa.descripcion.color`. Observe que el otro miembro de la unión, `camisa.descripcion.talla`, tendrá un valor sin sentido. Los valores de ambos miembros de la unión son entonces mostrados.

Luego asignamos el valor 12 a `camisa.descripcion.talla`, reescribiendo así el valor de `camisa.descripcion.color`. De nuevo se muestran a continuación los valores de ambos miembros de la unión.

La ejecución del programa produce la siguiente salida:

```
2
B -24713
@ 12
```

La primera línea indica que la unión se almacena en 2 bytes de memoria para guardar una cantidad entera. En la segunda línea, el primer dato (`B`) tiene sentido, pero no el segundo (`-24713`). En la tercera línea, el primer dato (`@`) no tiene sentido, pero el segundo sí lo tiene (`12`). Así, cada línea de salida tiene un valor con sentido de acuerdo con la instrucción de asignación que precede a cada instrucción `printf`.

Una variable de unión puede ser inicializada siempre que su tipo de almacenamiento sea `extern` o `static`. Recordar, sin embargo, que *sólo uno de los miembros de una unión puede*

tener un valor asignado en cada momento. La mayoría de los compiladores aceptarán un valor inicial para uno solo de los miembros de la unión y asignarán este valor al primer miembro dentro de la unión.

EJEMPLO 11.36. A continuación se muestra un sencillo programa en C que incluye la asignación de valores iniciales a una variable de estructura.

```
#include <stdio.h>

main()
{
   union id {
       char color[12];
       int talla;
   };

   struct ropa {
       char fabricante[20];
       float coste;
       union id descripcion;
   };

   static struct ropa camisa = {"Americana", 25.00, "blanca"};

   printf("%d\n", sizeof(union id));
   printf("%s %5.2f ", camisa.fabricante, camisa.coste);
   printf("%s %d\n", camisa.descripcion.color, camisa.descripcion.talla);

   camisa.descripcion.talla = 12;
   printf("%s %5.2f ", camisa.fabricante, camisa.coste);
   printf("%s %d\n", camisa.descripcion.color, camisa.descripcion.talla);
}
```

Observe que `camisa` es una variable de estructura del tipo `ropa`. Uno de sus miembros es `descripcion`, que es una unión del tipo `id`. Esta unión consta de dos miembros: un array de `12` caracteres y un entero.

La declaración de la variable de estructura incluye la asignación de los siguientes valores iniciales: `"Americana"` se asigna al miembro de array `camisa.fabricante`, `25.00` al miembro en coma flotante `camisa.coste` y `"blanca"` al miembro de la unión `camisa.descripcion.color`. Observe que el segundo miembro de la unión dentro de la estructura, `camisa.descripcion.talla`, queda sin especificar.

El programa muestra primero el tamaño del bloque de memoria reservado para la unión seguido por el valor de cada uno de los miembros de `camisa`. Luego se asigna `12` a `camisa.descripcion.talla` y se muestra de nuevo el valor de cada miembro de `camisa`.

Cuando se ejecuta el programa, se genera la siguiente salida:

```
12
Americana 25.00 blanca 26743
Americana 25.00 ~ 12
```

La primera línea indica que se reservan 12 bytes de memoria para la unión, para poder almacenar el array de 12 caracteres. La segunda línea muestra los valores asignados inicialmente a `camisa.fabricante`, `camisa.coste` y `camisa.descripcion.color`. El valor mostrado para `camisa.descripcion.talla` no tiene sentido. En la tercera línea vemos que `camisa.fabricante` y `camisa.coste` están sin modificar. Sin embargo, la reasignación de los miembros de la unión produce que el valor de `camisa.descripcion.color` no tenga sentido pero `camisa.descrip-ción.talla` muestre su nuevo valor 12.

En todos los otros aspectos, las uniones se procesan de la misma manera y con las mismas restricciones que las estructuras. Así, un miembro individual de una unión puede ser procesado como si fuera una variable ordinaria del mismo tipo de datos, y los punteros a uniones pueden ser pasados a o desde funciones (por referencia). Además, la mayoría de los compiladores de C permiten asignar una unión completa a otra unión, siempre que ambas uniones tengan la misma composición. Estos compiladores permiten también que uniones completas sean pasadas a o desde funciones (por valor), como se indica en el nuevo estándar ANSI.

EJEMPLO 11.37. Elevación de un número a una potencia. Este ejemplo es un poco complicado, pero ilustra cómo se puede usar una unión para pasar información a una función. El problema es elevar un número a una potencia. Queremos evaluar la fórmula $y = x^n$, donde x e y son valores en coma flotante y n puede ser un número o entero o en coma flotante.

Si n es entero, entonces y puede ser evaluado multiplicando x por sí misma un determinado número de veces. Por ejemplo, la cantidad x^3 puede ser expresada en términos del producto $(x)(x)(x)$. Si n es un valor en coma flotante, entonces se puede escribir $\log y = n \log x$, o $y = e^{(n \log x)}$. En el último caso x debe ser una cantidad positiva, ya que no se puede efectuar el logaritmo de cero o de una cantidad negativa.

Introduzcamos las siguientes declaraciones:

```
typedef  union  {
   float fexp;           /* exponente en coma flotante */
   int nexp;             /* exponente entero */
} nvalor;

typedef  struct  {
   float x;              /* valor para elevar a la potencia */
   char indicador;       /* 'f' si el exponente es en coma flotante
                            'e' si el exponente es entero */
   nvalor exp;           /* unión que contiene el exponente */
} valores;

valores a;
```

Así, `nvalor` es un tipo de unión definida por el usuario, consistente en el miembro en coma flotante `fexp` y el miembro entero `nexp`. Estos dos miembros representan los dos posibles tipos de exponentes en la expresión $y = x^n$. Análogamente, `valores` es un tipo de estructura definida por el usuario, que consta de un miembro en coma flotante `x`, un miembro carácter `indicador` y una unión del tipo `nvalor` llamada `exp`. Observe que `indicador` indica el tipo de exponente actualmente representado por la unión. Si indicador representa `'e'`, entonces la unión representará un exponente entero (`nexp` tendrá asignado un valor en ese momento); y si `indicador` representa `'f'`, entonces la unión representará un exponente en coma flotante (`fexp` tendrá asignado un valor en ese momento). Finalmente vemos que `a` es una variable de estructura del tipo `valores`.

Con estas declaraciones, es fácil escribir una función que evalúe la fórmula $y = x^n$, como sigue.

```
float potencia(valores a)   /* realiza la potenciación */

{
   int i;
   float y = a.x;

   if (a.indicador == 'e') {      /* exponente entero */
      if (a.exp.nexp == 0)
         y = 1.0;                 /* exponente cero */
      else {
         for (i = 1; i < abs(a.exp.nexp); ++i)
            y *= a.x;
         if (a.exp.nexp < 0)
            y = 1./y;             /* exponente entero no negativo */
      }
   }
   else                           /* exponente en coma flotante */
      y = exp(a.exp.fexp * log(a.x));

   return(y);
}
```

Esta función acepta una variable de estructura (a) del tipo `valores` como argumento. El método usado para realizar los cálculos depende del valor asignado a `a.indicador`. Si `a.indicador` tiene asignado el carácter `'e'`, entonces la potenciación se realiza multiplicando `a.x` por sí mismo un número apropiado de veces. En otro caso, la potenciación se realiza usando la fórmula $y = e^{(n \log x)}$. Observe que la función contiene correcciones para acomodar un exponente cero ($y = 1.0$) y exponentes enteros negativos.

Se añade a la función `main` la petición de los valores de x y n y se determina si n es un entero o no (comparando n con su valor truncado), se asigna un valor apropiado a `a.indicador` y `a.exp`, se llama a `potencia` y después se escribe el resultado. Se incluye también la previsión de generar un mensaje de error si n es un exponente en coma flotante y si el valor de x es menor o igual a cero.

Aquí tenemos el programa completo.

```
/* programa para elevar un número a una potencia */

#include <stdio.h>
#include <math.h>

typedef union {
    float fexp;              /* exponente en coma flotante */
    int nexp;                /* exponente entero */
} nvalor;

typedef struct {
    float x;                 /* valor para elevar a la potencia */
    char indicador;          /* 'f' si el exponente es en coma flotante
                                'e' si el exponente es entero */
    nvalor exp;              /* unión que contiene el exponente */
} valores;
```

```
float potencia(valores a); /* prototipo de función */

main()
{
    valores a;              /* estructura que contiene x, indica-
                               dor y fexp/nexp */
    int i;
    float n, y;

   /* introducción de datos de entrada */
   printf("y = x^n\n\nIntroducir un valor para x: ");
   scanf("%f", &a.x);
   printf("Introducir un valor para n: ");
   scanf("%f", &n);

   /* determinar el tipo de exponente */
   i = (int) n;
   a.indicador = (i == n) ? 'e' : 'f';
   if (a.indicador == 'e')
      a.exp.nexp = i;
   else
      a.exp.fexp = n;

   /* elevar x a la potencia adecuada y mostrar el resultado */
   if (a.indicador == 'f' && a.x <= 0.0) {
      printf("\nERROR - No se puede elevar un número no positivo a ");
      printf("una potencia en coma flotante");
   }
   else {
      y = potencia(a);
      printf("\ny = %.4f", y);
   }
}

float potencia(valores a)  /* realiza la potenciación */

{
   int i;
   float y = a.x;

   if (a.indicador == 'e') {          /* exponente entero */
      if (a.exp.nexp == 0)
         y = 1.0;                  /* exponente cero */
      else {
         for (i = 1; i < abs(a.exp.nexp); ++i)
            y *= a.x;
         if (a.exp.nexp < 0)
            y = 1./y;         /* exponente entero no negativo */
      }
   }
```

```
    else                          /* exponente en coma flotante */
        y = exp(a.exp.fexp * log(a.x));

    return(y);
}
```

Observe que las declaraciones de la unión y de la estructura son externas a las funciones del programa, pero la variable de estructura a se define localmente dentro de cada función.

Este programa no se ejecuta de modo repetitivo. Varios diálogos típicos representando distintas ejecuciones del programa se muestran a continuación. Como siempre, las respuestas del usuario están subrayadas.

```
Introducir un valor para x: 2
Introducir un valor para y: 3

y = 8.0000

Introducir un valor para x: -2
Introducir un valor para y: 3

y = -8.0000

Introducir un valor para x: 2.2
Introducir un valor para y: 3.3

y = 13.4895

Introducir un valor para x: -2.2
Introducir un valor para y: 3.3

ERROR - No se puede elevar un número no positivo a una potencia en
coma flotante
```

Debe señalarse que la mayoría de los compiladores de C incluyen la función de biblioteca pow, que se usa para elevar un número a una potencia. Se ha utilizado pow en varios ejemplos de programación anteriores (ver Ejemplos 5.2, 5.4, 6.21, 8.13 y 10.30). El programa actual no tiene sentido para sustituir a pow; se ha presentado sólo para ilustrar el uso de una unión en una situación representativa de programación.

CUESTIONES DE REPASO

11.1. ¿Qué es una estructura? ¿En qué se diferencia una estructura de un array?
11.2. ¿Qué es un miembro? ¿Cuál es la relación entre un miembro y una estructura?
11.3. Describir la sintaxis para definir una estructura. ¿Pueden ser inicializados los miembros individuales dentro de una declaración de tipo estructura?

11.4. ¿Cómo pueden declararse variables de estructura? ¿En qué se diferencian las declaraciones de variables de estructura de las declaraciones de tipos de estructuras?

11.5. ¿Qué es una marca? ¿Debe incluirse una marca en la declaración de una variable de estructura? Explicarlo.

11.6. ¿Se puede definir una variable de estructura como miembro de otra estructura? ¿Puede incluirse un array como miembro de una estructura? ¿Puede un array tener estructuras como elementos?

11.7. ¿Cómo se le asignan valores iniciales a los miembros de una estructura? ¿Qué restricciones se aplican a los tipos de almacenamiento de una estructura cuando se asignan valores iniciales?

11.8. ¿Cómo se inicializa un array de estructuras?

11.9. ¿Cuál es el ámbito de un nombre de miembro? ¿Qué implica esto con respecto al nombre de los miembros dentro de estructuras diferentes?

11.10. ¿Cómo se accede a un miembro de una estructura? ¿Cómo se puede procesar un miembro de una estructura?

11.11. ¿Cuál es la precedencia del operador punto (`.`)? ¿Cómo es su asociatividad?

11.12. ¿Puede usarse el operador punto con un array de estructuras? Explicarlo.

11.13. ¿Cuál es la única operación que se puede aplicar a la estructura completa en algunas versiones antiguas de C? ¿Cómo se ha modificado esta regla en las nuevas versiones del lenguaje compatibles con el estándar ANSI actual?

11.14. ¿Cómo se puede determinar el tamaño de una estructura? ¿En qué unidad se indica el tamaño?

11.15. ¿Cuál es la función de la característica `typedef`? ¿Cómo se usa esta característica conjuntamente con las estructuras?

11.16. ¿Cómo se declara una variable puntero a estructura? ¿A qué apunta este tipo de variable?

11.17. ¿Cómo se puede acceder a un miembro individual de una estructura en términos de su correspondiente variable puntero?

11.18. ¿Cuál es la precedencia del operador `->`? ¿Cómo es su asociatividad? Comparar las respuestas con las de la sección 11.11.

11.19. Supongamos que una variable puntero apunta a una estructura que contiene otra estructura como miembro. ¿Cómo puede ser accedido un miembro de la estructura interna?

11.20. Supongamos que una variable puntero apunta a una estructura que contiene un array como miembro. ¿Cómo puede ser accedido un elemento del array interno?

11.21. Supongamos que un miembro de una estructura es una variable puntero. ¿Cómo puede ser accedido el objeto del puntero en términos del nombre de la variable de estructura y del nombre del miembro?

11.22. ¿Qué ocurre cuando se incrementa un puntero a una estructura? ¿Qué peligro está asociado con este tipo de operación?

11.23. ¿Cómo se puede pasar una estructura completa a una función? Describirlo ampliamente, tanto para las antiguas como para las nuevas versiones de C.

11.24. ¿Cómo puede ser devuelta una estructura completa por una función? Describirlo ampliamente, tanto para las antiguas como para las nuevas versiones de C.

11.25. ¿Qué es una estructura autorreferenciadora? ¿Para qué tipo de aplicaciones son útiles las estructuras autorreferenciadoras?

11.26. ¿Cuál es la idea básica de una estructura de datos enlazada? ¿Qué ventajas tiene el uso de estructuras de datos enlazadas?

11.27. Indicar varios tipos de estructuras de datos enlazadas de uso común.

11.28. ¿Qué es una unión? ¿En qué se diferencia de una estructura?

11.29. ¿Para qué tipo de aplicaciones son útiles las uniones?

11.30. ¿En qué sentido se pueden mezclar las uniones, estructuras y arrays?

11.31. ¿Cómo se accede a un miembro de una unión? ¿Cómo puede procesarse un miembro de una unión? Comparar las respuestas con las de la Cuestión 11.10.

11.32. ¿Cómo se le asigna un valor inicial a un miembro de una variable de unión? ¿En qué se diferencia la inicialización de una variable de unión de la de una variable de estructura?

11.33. Resumir las reglas que se aplican al procesamiento de uniones. Comparar con las reglas que se aplican al procesamiento de estructuras.

PROBLEMAS

11.34. Definir una estructura que conste de dos miembros en coma flotante llamados `real` e `imaginario`. Incluir la marca `complejo` dentro de la definición.

11.35. Declarar las variables `x1`, `x2` y `x3` como estructuras del tipo `complejo`, descrita en el problema anterior.

11.36. Combinar la definición y la declaración de la estructura descritas en los Problemas 11.34 y 11.35 en una sola declaración.

11.37. Declarar una variable `x` como una estructura del tipo `complejo`, descrita en el Problema 11.34. Asignar los valores iniciales `1.3` y `-2.2` a los miembros `x.real` y `x.imaginario`, respectivamente.

11.38. Declarar una variable puntero, `px`, que apunte a una estructura del tipo `complejo`, descrita en el Problema 11.34. Escribir expresiones para los miembros de la estructura en términos de la variable puntero.

11.39. Declarar un array unidimensional de `100` elementos, llamado `cx`, cuyos elementos sean estructuras del tipo `complejo`, descrita en el Problema 11.34.

11.40. Combinar la declaración de la estructura y la definición del array descritas en los Problemas 11.34 y 11.39 en una sola declaración.

11.41. Supongamos que `cx` es un array unidimensional de `100` estructuras, como se describe en el Problema 11.39. Escribir expresiones para los miembros del elemento decimoctavo del array (elemento número `17`).

11.42. Definir una estructura que contenga los siguientes tres miembros:

a) una cantidad entera llamada `ganados`
b) una cantidad entera llamada `perdidos`
c) una cantidad en coma flotante llamada `porcentaje`

Incluir el tipo de dato definido por el usuario `registro` dentro de la definición.

11.43. Definir una estructura que contenga los dos miembros siguientes:

a) un array de `40` caracteres llamado `nombre`
b) una estructura llamada `posicion`, del tipo `registro` definido en el Problema 11.42.

Incluir el tipo de dato definido por el usuario `equipo` dentro de la definición.

11.44. Declarar una variable de estructura `t` del tipo `equipo`, descrito en el Problema 11.43. Escribir una expresión para cada miembro y cada submiembro de `t`.

11.45. Declarar una variable de estructura `t` del tipo `equipo`, como en el Problema anterior. Sin embargo, inicializar ahora `t` como sigue:

```
nombre:  Osos  de  Chicago
ganados:  14
perdidos:  2
porcentaje:  87.5
```

11.46. Escribir una instrucción que muestre el tamaño del bloque de memoria asociado con la variable `t` que fue descrita en el Problema 11.44.

11.47. Declarar una variable puntero `pt`, que apunte a una estructura del tipo `equipo`, descrita en el Problema 11.43. Escribir una expresión para cada miembro y submiembro de la estructura.

11.48. Declarar un array unidimensional de `48` elementos llamado `liga` cuyos elementos son estructuras del tipo `equipo`, descrito en el Problema 11.43. Escribir expresiones para el nombre y el porcentaje del quinto equipo en la liga (el equipo número 4).

11.49. Definir una estructura autorreferenciadora con los siguientes tres miembros:

a) un array de `40` caracteres llamado `nombre`
b) una estructura llamada `posicion` del tipo `registro`, descrito en el Problema 11.42
c) un puntero a otra estructura del mismo tipo, llamado `sig`
Incluir la marca `equipo` dentro de la definición de la estructura. Comparar su solución con la del Problema 11.43.

11.50. Declarar `pt` como un puntero a la estructura descrita en el problema anterior. Escribir una instrucción que reserve un bloque adecuado de memoria con el puntero `pt` apuntando al principio del bloque de memoria.

11.51. Definir una estructura del tipo `hms` que contenga tres miembros enteros, llamados `hora`, `minuto` y `segundo`, respectivamente. Después definir una unión con dos miembros, cada uno de ellos una estructura del tipo `hms`. Llamar a los miembros de la unión `local` y `hogar`, respectivamente. Declarar una variable puntero llamada `hora` que apunte a esta unión.

11.52. Definir una unión del tipo `res` que contenga los siguientes tres miembros:

a) una cantidad entera llamada `eres`
b) una cantidad en coma flotante llamada `fres`
c) una cantidad en doble precisión llamada `dres`

Después definir una estructura que contenga los siguientes cuatro miembros:

a) una unión del tipo `res` llamada `respuesta`
b) un carácter llamado `indicador`
c) cantidades enteras llamadas `a` y `b`
Finalmente declarar dos variables de estructura, llamadas `x` e `y`, cuya composición sea la descrita anteriormente.

11.53. Declarar una variable de estructura `v` cuya composición está descrita en el Problema 11.52. Asignar los siguientes valores iniciales dentro de la declaración:

```
respuesta:  14
indicador:  'e'
a:  -2
b:  5
```

11.54. Modificar la definición de la estructura descrita en el Problema 11.52 de modo que contenga un miembro adicional, llamado `sig`, que es un puntero a otra estructura del mismo tipo. (Observe que la estructura ahora será autorreferenciadora.) Añadir una declaración de dos variables, llamadas `x` y `px`, donde `x` es una variable de estructura y `px` un puntero a una variable de estructura. Asignar la dirección de comienzo de `x` a `px` dentro de la declaración.

11.55. Describir la salida generada por cada uno de los siguientes programas. Explicar cualquier diferencia entre ellos.

a)
```
#include <stdio.h>

typedef struct {
     char *a;
     char *b;
     char *c;
} colores;

void func(colores muestra);

main()
{
     static colores muestra = {"rojo", "verde", "azul"};

     printf("%s %s %s\n", muestra.a, muestra.b, muestra.c);
     func(muestra);
     printf("%s %s %s\n", muestra.a, muestra.b, muestra.c);
}

void func(colores muestra)
{
     muestra.a = "cian";
     muestra.b = "magenta";
     muestra.c = "amarillo";
     printf("%s %s %s\n", muestra.a, muestra.b, muestra.c);
     return;
}
```

b)
```
#include <stdio.h>

typedef struct {
     char *a;
     char *b;
     char *c;
} colores;

void func(colores *pt);
```

```
main()
{
     static colores muestra = {"rojo", "verde", "azul"};

     printf("%s %s %s\n", muestra.a, muestra.b, muestra.c);
     func(&muestra);
     printf("%s %s %s\n", muestra.a, muestra.b, muestra.c);
}

void func(colores *pt)
{
     pt->a = "cian";
     pt->b = "magenta";
     pt->c = "amarillo";
     printf("%s %s %s\n", pt->a, pt->b, pt->c);
     return;
}
```

c)

```
#include <stdio.h>

typedef struct {
     char *a;
     char *b;
     char *c;
} colores;

colores func(colores muestra);

main()
{
     static colores muestra = {"rojo", "verde", "azul"};

     printf("%s %s %s\n", muestra.a, muestra.b, muestra.c);
     muestra = func(muestra);
     printf("%s %s %s\n", muestra.a, muestra.b, muestra.c);
}

colores func(colores muestra)
{
     muestra.a = "cian";
     muestra.b = "magenta";
     muestra.c = "amarillo";
     printf("%s %s %s\n", muestra.a, muestra.b, muestra.c);
     return(muestra);
}
```

11.56. Describir la salida generada por el siguiente programa. Distinguir entre salidas con y sin sentido.

```
#include <stdio.h>

main()
{
```

```
        union {
            int i;
            float f;
            double d;
        } u;

        printf("%d\n", sizeof u);
        u.i = 100;
        printf("%d %f %f\n", u.i, u.f, u.d);
        u.f = 0.5;
        printf("%d %f %f\n", u.i, u.f, u.d);
        u.d = 0.016667;
        printf("%d %f %f\n", u.i, u.f, u.d);
    }
```

11.57. Describir la salida generada por cada uno de los siguientes programas. Explicar cualquier diferencia entre ellos.

a)
```
#include <stdio.h>

typedef union {
     int i;
     float f;
} udef;

void func(udef u);

main()
{
     udef u;

     u.i = 100;
     u.f = 0.5;
     func(u);
     printf("%d %f\n", u.i, u.f);
}

void func(udef u)
{
     u.i = 200;
     printf("%d %f\n", u.i, u.f);
     return;
}
```

b)
```
#include <stdio.h>

typedef union {
     int i;
     float f;
} udef;

void func(udef u);
```

```
main()
{
     udef u;

     u.i = 100;
     u.f = 0.5;
     func(u);
     printf("%d %f\n", u.i, u.f);
}

void func(udef u)
{
     u.f = -0.3;
     printf("%d %f\n", u.i, u.f);
     return;
}
```

c)
```
#include <stdio.h>

typedef union {
     int i;
     float f;
} udef;

udef func(udef u);

main()
{
     udef u;

     u.i = 100;
     u.f = 0.5;
     u = func(u);
     printf("%d %f\n", u.i, u.f);
}

udef func(udef u)
{
     u.f = -0.3;
     printf("%d %f\n", u.i, u.f);
     return(u);
}
```

PROBLEMAS DE PROGRAMACIÓN

11.58. Contestar a las siguientes cuestiones que se refieren a su compilador o intérprete particular de C.

a) ¿Puede ser asignada una variable de estructura (o variable de unión) a otra variable de estructura (unión), suponiendo que ambas variables tienen la misma composición?

b) ¿Se puede pasar una variable de estructura (o de unión) a una función como argumento?
c) ¿Puede ser devuelta una variable de estructura completa (o de unión) desde una función hasta la rutina que hizo la llamada?
d) ¿Puede pasarse un puntero a estructura (o unión) a una función como un argumento?
e) ¿Puede devolverse un puntero a estructura (o unión) desde una función hasta la rutina que hizo la llamada?

11.59. Modificar el programa dado en el Ejemplo 11.26 (localización de registros de clientes) de modo que la función `buscar` devuelva una estructura completa en vez de un puntero a una estructura. (No intentar este problema si su versión de C no soporta la devolución de estructuras completas desde una función.)

11.60. Modificar el programa de facturación mostrado en el Ejemplo 11.28 de modo que cualquiera de los siguientes informes pueda ser impreso:

a) Estado de todos los clientes (ahora generado por el programa)
b) Estado sólo de los clientes atrasados y delincuentes
c) Estado sólo de los clientes delincuentes

Incluir la posibilidad de que cuando el programa se ejecute se muestre un menú en el cual el usuario pueda elegir el tipo de informe a generar. El programa vuelve al menú después de imprimir el informe, permitiendo así la posibilidad de generar varios informes distintos.

11.61. Modificar el programa de facturación mostrado en el Ejemplo 11.28 de modo que la estructura del tipo `registro` incluya ahora una unión que contenga los miembros `direc_oficina` y `direc_casa`. Cada miembro de la unión debe ser a su vez una estructura que consta de dos arrays de `80` caracteres llamados `calle` y `ciudad`, respectivamente. Añadir otro miembro a la estructura primaria (del tipo `registro`), que sea un carácter llamado `indicador`. Este miembro debe tener asignado un carácter (`'o'` o `'c'`) para indicar qué tipo de dirección está actualmente almacenada en la unión.

Modificar el resto del programa de modo que se pregunte al usuario qué tipo de dirección será dada para cada cliente. Después se muestra la dirección apropiada con su correspondiente etiqueta, conjuntamente con el resto de la salida.

11.62. Modificar el programa dado en el Ejemplo 11.37 de modo que el número elevado a la potencia en coma flotante pueda ser ejecutado en simple o en doble precisión, según especifique el usuario una pregunta. El tipo de unión `nvalor` debe contener ahora un tercer miembro, que será una cantidad en doble precisión llamada `dexp`.

11.63. Reescribir cada uno de los siguientes programas en C de modo que haga uso de variables de estructura:

a) El programa de depreciación presentado en el Ejemplo 7.20.
b) El programa dado en el ejemplo 10.28 para mostrar el día del año.
c) El programa para determinar el valor futuro de una serie de depósitos mensuales, dado en el Ejemplo 10.31.

11.64. Modificar el generador de «pig latin» presentado en el Ejemplo 9.14 de modo que acepte múltiples líneas de texto. Representar cada línea de texto con una estructura separada. Incluir los siguientes tres miembros en cada estructura:

a) La línca dc texto original
b) El número de palabras dentro de la línea
c) La línea de texto modificada (el «pig latin» equivalente del texto original)
Incluir las mejoras descritas en el Problema 9.36 (marcas de puntuación, letras mayúsculas y sonidos de letras dobles).

11.65. Escribir un programa en C que lea varios nombres y direcciones, reordene los nombres alfabéticamente y escriba la lista alfabética. (Ver Ejemplos 9.20 y 10.26.) Hacer uso de variables de estructura dentro del programa.

11.66. Escribir un programa completo en C que haga uso de variables de estructura para cada uno de los siguientes problemas de programación descritos en los capítulos anteriores.

a) La calificación media de los estudiantes, descrito en el Problema 9.40.
b) La versión más comprensible de la calificación media de los estudiantes, descrita en el Problema 9.42.
c) El problema que relaciona los nombres de los países con sus correspondientes capitales, descrito en el Problema 9.46.
d) El problema de codificar-decodificar texto descrito en el Problema 9.49, pero extendido a múltiples líneas de texto.

11.67. Escribir un programa en C que acepte la siguiente información para cada equipo de la liga de béisbol o fútbol:

1. Nombre del equipo, incluida la ciudad (por ejemplo, Pittsburgh Steelers)
2. Número de victorias
3. Número de derrotas

Para un equipo de béisbol, añadir la siguiente información:

4. Número de bolas bateadas con éxito
5. Número de carreras
6. Número de errores
7. Número de juegos extra

Análogamente, añadir la siguiente información para un equipo de fútbol:

4. Número de empates
5. Número de tantos
6. Número de goles de campo
7. Número de contraataques
8. Total de yardas ganadas (total de la temporada)
9. Total de yardas cedidas a los oponentes

Introducir esta información para todos los equipos de la liga. Después reordenar y escribir la lista de equipos de acuerdo con su registro de victorias-derrotas, usando las técnicas de reordenación descritas en los Ejemplos 9.13 y 10.16 (ver también los Ejemplos 9.21 y 10.26). Almacenar la información en un array de estructuras, donde cada elemento del array (cada estructura) contiene la información para un equipo. Hacer uso de una unión para representar la información variable (o béisbol o fútbol) que se incluye como parte de la estructura. La unión

debe a su vez contener dos estructuras, una para las estadísticas relacionadas con el béisbol y otra para las estadísticas del fútbol.

Comprobar el programa usando las estadísticas de la temporada actual. (Idealmente, el programa debe ser comprobado usando estadísticas de béisbol y de fútbol.)

11.68. Modificar el programa dado en el Ejemplo 11.32 de modo que haga uso de cada una de las siguientes estructuras enlazadas:

a) Una lista lineal enlazada con dos conjuntos de punteros: uno apuntando hacia delante y el otro apuntando hacia detrás.
b) Una lista enlazada circular. Se debe incluir un puntero que identifique el principio de la lista circular.

11.69. Modificar el programa dado en el Ejemplo 11.32 de modo que cada nodo contenga la siguiente información:

a) Nombre
b) Calle
c) Ciudad/Estado/Código
d) Número de cuenta
e) Estado de la cuenta (un carácter indicando al día, atrasada o delincuente)

11.70. Escribir un programa completo en C que permita introducir y mantener una versión computarizada del árbol familiar. Empezar especificando el número de generaciones (el número de niveles dentro del árbol). Después introducir los nombres y nacionalidades de forma jerárquica, empezando con su propio nombre y nacionalidad. Incluir la posibilidad de modificar el árbol y de añadir nuevos nombres (nuevos nodos) al árbol. También introducir la posibilidad de mostrar el árbol completo automáticamente después de cada puesta al día.

Comprobar el programa incluyendo al menos tres generaciones si es posible (usted, sus padres y sus abuelos). Obviamente, el árbol es más interesante cuando el número de generaciones aumenta.

11.71. Una calculadora RPN utiliza un esquema donde cada nuevo valor numérico es seguido por la operación que se realizará entre el nuevo valor y su predecesor. (RPN proviene de «notación polaca invertida», en inglés *Reverse Polish Notation*.) De esta forma, sumar dos números, tales como `3.3` y `4.8`, requerirá las siguientes pulsaciones:

```
3.3 <intro>
4.8 +
```

La suma, `8.1`, se mostrará en el registro visible de la calculadora.

La calculadora RPN hace uso de una *pila*, que contiene típicamente cuatro registros (cuatro componentes), como se ilustra en la Figura 11.7. Cada nuevo número se introduce en el registro `X`, haciendo que los anteriores valores introducidos sean «empujados hacia arriba» en la pila. Si el registro superior (el registro `T`) estaba previamente ocupado, entonces el número antiguo se perderá (será reescrito por el valor empujado desde el registro `Z`).

Las operaciones aritméticas son siempre realizadas entre los números de los registros `X` e `Y`. El resultado de tal operación será siempre mostrado en el registro `X`, haciendo que los registros superiores bajen un nivel («sacando» de la pila). Este procedimiento se ilustra en la Figura 11.8(*a*) a (*c*) para la suma de los valores `3.3` y `4.8`, como se describió anteriormente.

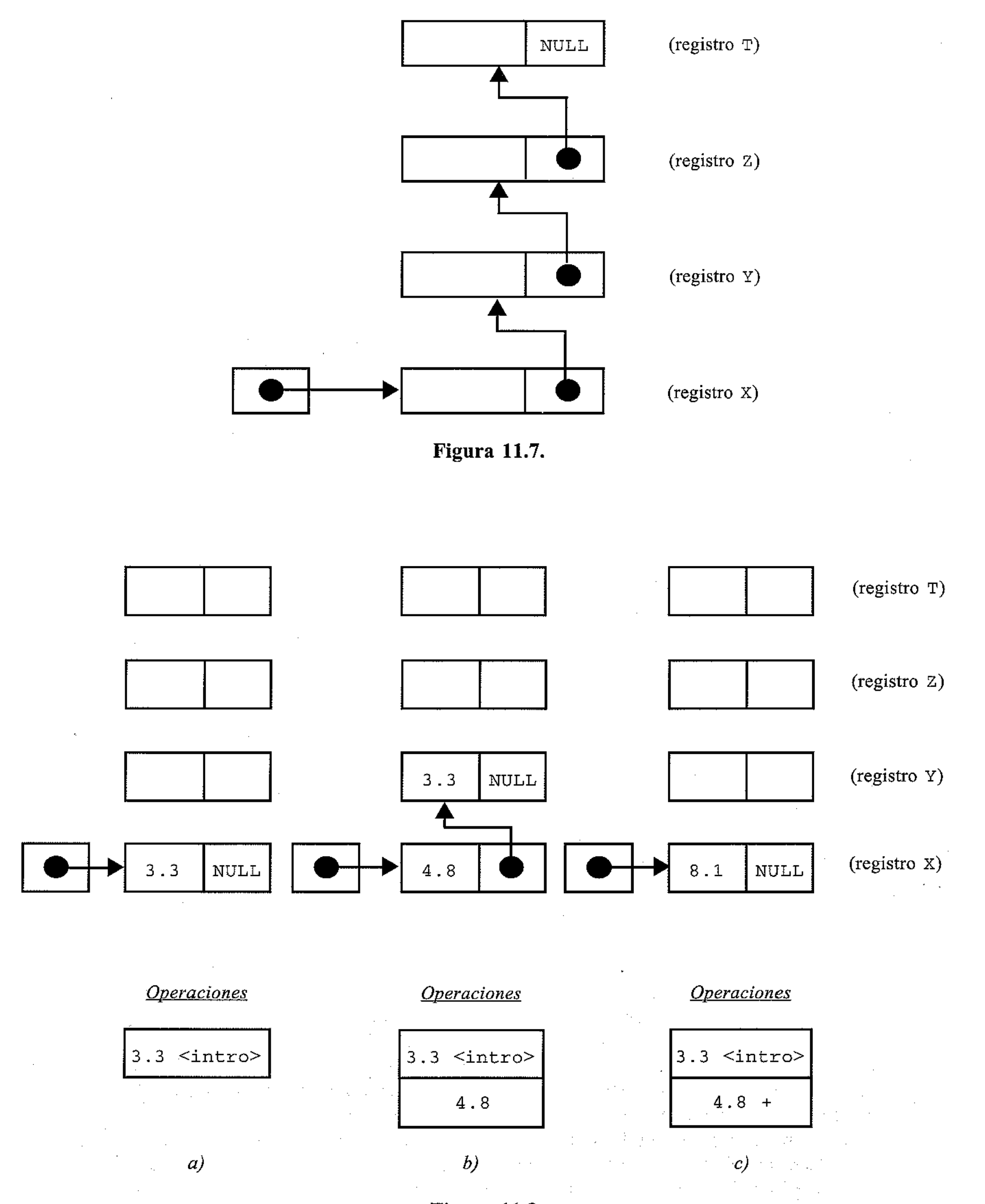


Figura 11.7.

Figura 11.8.

Escribir un programa interactivo en C que simule una calculadora RPN. Mostrar los contenidos de la pila tras cada operación, como en la Figura 11.8(*a*) a (*c*). Incluir la posibilidad de realizar cada una de las siguientes operaciones.

Operación	*Pulsaciones*
introducir un nuevo dato	`(valor) <intro>`
suma	`(valor) +`
resta	`(valor) -`
multiplicación	`(valor) *`
división	`(valor) /`

Comprobar el programa usando datos numéricos de su elección.

CAPÍTULO 12

Archivos de datos

Muchas aplicaciones requieren escribir o leer información de un dispositivo de almacenamiento auxiliar. Tal información se guarda en el dispositivo de almacenamiento en la forma de un *archivo de datos*. Por tanto, los archivos de datos nos permiten almacenar información de modo permanente y acceder y modificar la misma cuando sea necesario.

En C existe un conjunto amplio de funciones de biblioteca para crear y procesar archivos de datos. A diferencia de otros lenguajes de programación, en C no se distingue entre archivos secuenciales y de acceso directo (acceso aleatorio). Pero existen dos tipos diferentes de archivos de datos, llamados archivos *secuenciales* de datos (o *estándar*) y archivos *orientados al sistema* (o *de bajo nivel*). Generalmente es más fácil trabajar con archivos de datos secuenciales que con los orientados al sistema, y por tanto son los más usados.

Los archivos de datos secuenciales se pueden dividir en dos categorías. En la primera categoría están los archivos de *texto*, que contienen caracteres consecutivos. Estos caracteres pueden interpretarse como datos individuales, como componentes de una cadena de caracteres o como números. La manera de interpretarlos es determinada por las funciones de biblioteca usadas para transferir la información o por las especificaciones de formato dentro de las funciones de biblioteca, tales como `printf` o `scanf`, descritas en el Capítulo 4.

La segunda categoría de archivos de datos secuenciales, a menudo llamados archivos *sin formato*, organiza los datos en bloques de bytes contiguos de información. Estos bloques representan estructuras de datos más complejas, como arrays y estructuras. Hay un conjunto diferenciado de funciones de biblioteca para tratar este tipo de archivos. Estas funciones proveen instrucciones simples que pueden transferir arrays o estructuras completas a o desde un archivo de datos.

Los archivos orientados al sistema están más relacionados con el sistema operativo de la computadora que los archivos secuenciales. Es algo más complicado trabajar con ellos, pero su uso puede ser más eficiente para cierto tipo de aplicaciones. Para procesar archivos orientados al sistema se requiere un conjunto separado de procedimientos con sus funciones de biblioteca correspondientes.

Este capítulo trata sólo con archivos secuenciales. El enfoque general es prácticamente estándar, si bien los detalles pueden variar de una versión de C a otra. Así, los ejemplos presentados en este capítulo pueden no ser directamente aplicables a todas las versiones del lenguaje tal como se muestran. No obstante, deben tenerse pocas dificultades para acondicionar este material a su versión particular de C.

12.1. APERTURA Y CIERRE DE UN ARCHIVO

Cuando se trabaja con archivos secuenciales, el primer paso es establecer un *área de buffer*, donde la información se almacena temporalmente mientras se está transfiriendo entre la memoria de la computadora y el archivo de datos. Este área de buffer permite leer y escribir información del archivo más rápidamente de lo que sería posible de otra manera. El área de búffer se establece escribiendo

```
FILE *ptvar;
```

donde `FILE` (se requieren letras mayúsculas) es un tipo especial de estructura que establece el área de búffer y *`ptvar`* es la variable puntero que indica el principio de este área. El tipo de estructura `FILE` está definido en un archivo `include` del sistema, normalmente en `stdio.h`. El puntero *`ptvar`* se refiere a menudo como un *puntero a archivo secuencial*, o simplemente como *archivo secuencial.*

Un archivo debe ser *abierto* antes de ser creado o procesado. Esto asocia el nombre del archivo con el área de búffer (con el archivo secuencial). También se especifica cómo se va a usar el archivo, sólo para lectura, sólo para escritura, o para lectura/escritura, en el que se permiten ambas operaciones.

Para abrir un archivo se usa la función de biblioteca `fopen`. Esta función se escribe típicamente como

```
ptvar = fopen(nombre-archivo, tipo-archivo);
```

donde *`nombre-archivo`* y *`tipo-archivo`* son cadenas de caracteres que representan, respectivamente, el nombre del archivo y la manera en la que el archivo será utilizado. El nombre elegido para *`nombre-archivo`* debe ser consistente con las reglas para nombrar archivos del sistema operativo. El *`tipo-archivo`* debe ser una de las cadenas mostradas en la Tabla 12.1.

Tabla 12.1. Especificaciones de tipo de archivo

Tipo-archivo	*Significado*
`"r"`	Abrir un archivo existente sólo para lectura.
`"w"`	Abrir un nuevo archivo sólo para escritura. Si existe un archivo con el *`nombre-archivo`* especificado, será destruido y creado uno nuevo en su lugar.
`"a"`	Abrir un archivo existente para añadir (agregar información nueva al final del archivo). Se creará un archivo nuevo si no existe un archivo con el *`nombre-archivo`* especificado.
`"r+"`	Abrir un archivo existente tanto para lectura como para escritura.
`"w+"`	Abrir un archivo nuevo para lectura y escritura. Si existe un archivo con *`nombre-archivo`* especificado, será destruido y creado uno nuevo en su lugar.
`"a+"`	Abrir un archivo existente para leer y añadir. Se creará un nuevo archivo si no existe un archivo con el *`nombre-archivo`* especificado.

La función fopen devuelve un puntero al principio del área de buffer asociada con el archivo. Se devuelve un valor NULL si no se puede abrir el archivo, por ejemplo si no se puede encontrar un archivo existente.

Finalmente, un archivo de datos debe *cerrarse* al final del programa. Esto puede realizarse con la función de biblioteca fclose. La sintaxis es simplemente

```
fclose(ptvar);
```

Es una buena práctica de programación cerrar explícitamente los archivos de datos mediante la función fclose, aunque la mayoría de los compiladores de C cerrarán automáticamente los archivos de datos al final de la ejecución del programa si no está presente una llamada a fclose.

EJEMPLO 12.1. Un programa en C contiene las siguientes instrucciones:

```
#include <stdio.h>

FILE *fpt;

fpt = fopen("muestra.dat", "w");
. . . . .
fclose(fpt);
```

La primera instrucción incluye el archivo de cabecera stdio.h en el programa. La segunda instrucción define un puntero llamado fpt que apuntará a la estructura del tipo FILE, indicando el principio del área de buffer del archivo. Observe que FILE está definido en stdio.h.

La tercera instrucción abre un archivo nuevo llamado muestra.dat como un archivo de sólo escritura. Además, la función fopen devuelve un puntero al principio del área de buffer y lo asigna a la variable puntero fpt. Así, fpt apunta al área de buffer asociada con el archivo de datos muestra.dat. Todas las subsiguientes instrucciones de proceso (que no se muestran explícitamente en este ejemplo) accederán al archivo mediante el puntero fpt en vez de mediante el nombre del archivo.

Finalmente, la última instrucción cierra el archivo de datos. Observe que el argumento es la variable puntero fpt, no el nombre muestra.dat.

El valor devuelto por la función fopen puede usarse para generar un mensaje de error si el archivo de datos no puede abrirse, como se ilustra en el siguiente ejemplo.

EJEMPLO 12.2. Un programa en C contiene las siguientes instrucciones:

```
#include <stdio.h>
#define NULL 0

main()
{
  FILE *fpt;

  fpt = fopen("muestra.dat", "r+");

  if (fpt == NULL)
     printf("\nERROR - No se puede abrir el archivo indicado\n");
  else {
     . . . . .
     fclose(fpt);
  }
}
```

Este programa trata de abrir para lectura y escritura un archivo de datos previamente existente llamado `muestra.dat`. Se generará un mensaje de error si no se encuentra este archivo. En otro caso, el archivo de datos se abrirá y procesará como se indica.

Las instrucciones `fopen` e `if` se combinan frecuentemente como sigue:

```
if ((fpt = fopen("muestra.dat", "r+")) == NULL)
   printf("\nERROR - No se puede abrir el archivo indicado\n");
```

Cualquiera de los métodos es aceptable.

12.2. CREACIÓN DE UN ARCHIVO

Un archivo debe crearse antes de ser procesado. Un archivo *secuencial* puede crearse de dos formas distintas. Una es crear el archivo directamente, usando un procesador de texto o un editor. La otra es escribir un programa que introduzca información en la computadora y la escriba en un archivo. Los archivos *sin formato* sólo pueden crearse con programas especialmente escritos para tal fin.

Cuando se crea un nuevo archivo con uno de estos programas, el enfoque normal es introducir la información desde el teclado y escribirla en el archivo. Si el archivo consta de caracteres individuales, se pueden usar las funciones de biblioteca `getchar` y `putc` para introducir los datos desde el teclado y escribirlos en el archivo. Ya se ha discutido el uso de `getchar` en la sección 4.2. La función `putc` es nueva, pero su uso es análogo al de `putchar`, discutido en la sección 4.3.

EJEMPLO 12.3. Creación de un archivo de datos (conversión del texto de minúsculas a mayúsculas). Aquí tenemos una variación de algunos ejemplos anteriores, que leen una línea en minúsculas y la escriben en mayúsculas (ver Ejemplos 4.4, 6.9, 6.12, 6.16 y 9.2). En este ejemplo se lee el texto carácter a carácter usando la función `getchar` y después se escribe en un archivo usando `putc`. La función de biblioteca `toupper` realiza la conversión de minúsculas a mayúsculas, como antes.

El programa comienza definiendo el puntero a archivo secuencial `fpt`, que indica el principio del área de buffer del archivo. Se abre un nuevo archivo, `muestra.dat`, sólo para escritura. A continuación, el bucle `do - while` lee una serie de caracteres del teclado y escribe las mayúsculas equivalentes en el archivo. La función `putc` se usa para escribir cada carácter en el archivo. Observe que `putc` requiere que el puntero del archivo secuencial `fpt` sea especificado como argumento.

El bucle continúa hasta que se introduce un carácter de nueva línea (`'\n'`) desde el teclado. Cuando se detecta el carácter de nueva línea, se sale del bucle y se cierra el archivo.

```
/* leer una línea de texto en minúsculas y almacenarla en mayús-
   culas en un archivo de datos */

#include <stdio.h>
#include <ctype.h>

main()
{
```

```
    FILE *fpt;     /* define un puntero al tipo de estructura prede-
                      finida FILE */

    char c;

    /* abrir un archivo nuevo solo para escritura */
    fpt = fopen("muestra.dat", "w");

    /* leer cada carácter y escribir su mayúscula correspondiente
       en el archivo */
    do

         putc(toupper(c = getchar()), fpt);
    while (c != '\n');

    /* cerrar el archivo de datos */
    fclose(fpt);
}
```

Después de ejecutar el programa, el archivo `muestra.dat` contendrá las mayúsculas equivalentes a la línea introducida desde el teclado. Por ejemplo, si la línea original de texto ha sido

```
Nosotros, el pueblo de los Estados Unidos
```

el archivo contendría el texto

```
NOSOTROS, EL PUEBLO DE LOS ESTADOS UNIDOS
```

Un archivo creado de esta manera puede ser visualizado de distintas formas. Por ejemplo, el archivo puede ser visualizado directamente, usando una orden del sistema operativo tal como `print` o `type`. También se puede visualizar usando un editor o un procesador de texto.

Otra posibilidad es escribir un programa que lea el contenido del archivo y lo muestre. Este programa sería, en algún sentido, una imagen especular del descrito anteriormente; la función de biblioteca `getc` leerá caracteres individuales del archivo de datos y `putchar` los mostrará en la pantalla. Ésta es una forma más complicada de visualizar un archivo, pero ofrece gran flexibilidad, ya que los elementos individuales pueden procesarse según se leen.

EJEMPLO 12.4. Lectura de un archivo de datos. El siguiente programa lee una línea de texto de un archivo de datos, carácter a carácter, y muestra el texto en la pantalla. El programa hace uso de las funciones de biblioteca `getc` y `putchar` (ver sección 4.3) para leer y mostrar los datos. Complementa al programa presentado en el Ejemplo 12.3.

La lógica es análoga a la del programa mostrado en el Ejemplo 12.3. Sin embargo, observe que este programa abre el archivo `muestra.dat` sólo para lectura, mientras que el programa anterior lo abría sólo para escritura. Si el archivo `muestra.dat` no se puede abrir, se genera un mensaje de error. Finalmente, observe que `getc` requiere que el puntero al archivo secuencial `fpt` sea especificado como argumento.

```
/* leer una línea de texto de un archivo y mostrarla en la pantalla */

#include <stdio.h>
```

```
#define NULL 0

main()
{
  FILE *fpt;    /* define un puntero al tipo de estructura prede-
                   finida FILE */

  char c;

  /* abrir el archivo de datos solo para lectura */
  if ((fpt = fopen("muestra.dat", "r")) == NULL)
     printf("\nERROR - No se puede abrir el archivo indicado\n");

  else  /* leer y mostrar cada carácter del archivo */
     do
        putchar(c = getc(fpt));
     while (c != '\n');

  /* cerrar el archivo de datos */
  fclose(fpt);
}
```

Los archivos de datos que contienen sólo cadenas de caracteres pueden crearse y leerse más fácilmente con programas que utilizan funciones de biblioteca especialmente orientadas para cadenas de caracteres. Algunas funciones de este tipo comúnmente usadas son `gets`, `puts`, `fgets` y `fputs`. Las funciones `puts` y `gets` leen o escriben cadenas de caracteres a o desde los dispositivos de salida estándar, mientras que `fgets` y `fputs` intercambian cadenas con archivos. Como el uso de estas funciones es directo, no continuaremos más allá con este tema. Sin embargo, debería experimentar con estas funciones, rehaciendo algunos de los programas de lectura/escritura orientados a caracteres presentados previamente.

Muchos archivos de datos contienen estructuras de datos más complicadas, tales como registros que incluyen combinaciones de información numérica y de carácter. Tales archivos se pueden procesar usando las funciones de biblioteca `fscanf` y `fprintf`, que son análogas a las funciones `scanf` y `printf` discutidas en el Capítulo 4 (ver secciones 4.4 y 4.6). Así, la función `fscanf` permite leer un dato con formato desde un archivo de datos asociado con un archivo secuencial, y `fprintf` permite escribir datos con formato en el archivo de datos. Las especificaciones de formato son las mismas que se usan con las funciones `printf` y `scanf`.

EJEMPLO 12.5. Creación de un archivo que contiene registros de clientes. En el último capí-tulo se presentaron tres programas que eran usados supuestamente para crear y actualizar registros de clientes (ver Ejemplos 11.14 y 11.28). Cuando se describieron los programas se dijo que no eran reales, ya que se deberían usar archivos para esas aplicaciones. Ahora fijamos la atención en un programa que crea un archivo para una serie de registros de clientes cuya composición es como sigue:

```
typedef struct {
    int mes;
    int dia;
    int anio;
} fecha;
```

```
typedef struct {
     char nombre[80];
     char calle[80];
     char ciudad[80];
     int num_cuenta;
     int tipo_cuenta;
     float anteriorsaldo;
     float nuevosaldo;
     float pago;
     fecha ultimopago;
} registro;
```

La estrategia general será dar la fecha actual y entrar en un bucle que procesará una serie de registros de clientes. Para cada cliente se leerán el nombre del cliente, la calle, la ciudad, el número de cuenta (`num_cuenta`) y el saldo anterior (`anteriorsaldo`). A continuación se asignará el valor inicial 0 a los miembros de la estructura `nuevosaldo` y `pago`, se asignará el carácter `'A'` a `tipo_cuenta` (indicando el estado «Al día») y la fecha actual se asigna a `ultimopago`. Cada registro de cliente se escribirá en un archivo de sólo escritura llamado `registro.dat`.

El proceso continuará hasta que el nombre de un cliente comience con los caracteres `FIN` (tanto en mayúsculas como en minúsculas). Cuando se encuentre `FIN`, se escribirá en el archivo de datos indicando una condición de fin de archivo.

A continuación se muestra el programa completo.

```
/* crear un archivo de datos que contiene registros de clientes */

#include <stdio.h>
#include <string.h>

#define VERDADERO 1

typedef struct {
     int mes;
     int dia;
     int anio;
} fecha;

typedef struct {
     char nombre[80];
     char calle[80];
     char ciudad[80];
     int num_cuenta;                          /*(entero positivo) */
     int tipo_cuenta;                         /*A (Al día), R (atrasa-
                                                da) o D (delincuente) */
     float anteriorsaldo;                     /*(cantidad no negativa) */
     float nuevosaldo;                        /*(cantidad no negativa) */
     float pago;                              /*(cantidad no negativa) */
     fecha ultimopago;
} registro;

registro leerpantalla(registro cliente); /*prototipo de función */
void escribirarchivo(registro cliente);  /*prototipo de función */
```

```
FILE *fpt;            /* puntero a la estructura predefinida FILE */

main()
{
   int indicador = VERDADERO; /* declaración de variable */
   registro cliente;          /* declaración de variable de estructura */

   /* abrir un archivo nuevo solo para escritura */
   fpt = fopen("registro.dat", "w");

   /* introducir fecha y asignar valores iniciales */
   printf("SISTEMA DE FACTURACION DE CLIENTES - INICIALIZACION\n\n");
   printf("Introduzca la fecha actual (mm/dd/aaaa): ");
   scanf("%d/%d/%d", &cliente.ultimopago.mes,
                     &cliente.ultimopago.dia,
                     &cliente.ultimopago.anio);
   cliente.nuevosaldo = 0;
   cliente.pago = 0;
   cliente.tipo_cuenta = 'A';

   /* bucle principal */
   while (indicador) {
            /* introducir el nombre del cliente y escribirlo en el
               archivo */
            printf("\nNombre (introducir \'FIN\' para terminar): ");
            scanf(" %[^\n]", cliente.nombre);
            fprintf(fpt, "\n%s\n", cliente.nombre);

            /* comprobación de la condición de parada */
            if (strcmp(cliente.nombre, "FIN") == 0)
               break;

            cliente = leerpantalla(cliente);
            escribirarchivo(cliente);
   }

   fclose(fpt);
}

registro leerpantalla(registro cliente)
                                      /* leer el resto de datos */

{
   printf("Calle: ");
   scanf(" %[^\n]", cliente.calle);
   printf("Ciudad: ");
   scanf(" %[^\n]", cliente.ciudad);
   printf("Número de cuenta: ");
   scanf("%d", &cliente.num_cuenta);
   printf("Saldo actual: ");
```

```
    scanf("%f", &cliente.anteriorsaldo);
    return(cliente);
}

void escribirarchivo(registro cliente) /* escribir el resto de
                                           los datos en el archivo */
{
    fprintf(fpt, "%s\n", cliente.calle);
    fprintf(fpt, "%s\n", cliente.ciudad);
    fprintf(fpt, "%d\n", cliente.num_cuenta);
    fprintf(fpt, "%c\n", cliente.tipo_cuenta);
    fprintf(fpt, "%.2f\n", cliente.anteriorsaldo);
    fprintf(fpt, "%.2f\n", cliente.nuevosaldo);
    fprintf(fpt, "%.2f\n", cliente.pago);
    fprintf(fpt, "%d/%d/%d\n", cliente.ultimopago.mes,
                               cliente.ultimopago.dia,
                               cliente.ultimopago.anio);
    return;
}
```

El programa comienza definiendo la composición de cada registro de cliente y del puntero de archivo `fpt`. Dentro de `main` se abre un nuevo archivo de sólo escritura, llamado `registro.dat`. A continuación, el programa pide la fecha actual y se asignan valores iniciales a los miembros de la estructura `nuevosaldo`, `pago` y `tipo_cuenta`.

El programa entra en un bucle `while` que pide el nombre de un cliente y lo escribe en el archivo de datos. A continuación, el programa comprueba si el nombre introducido es `FIN` (mayúsculas o minúsculas). Si es así, se sale del bucle, se cierra el archivo y termina la ejecución. En otro caso, el resto de la información del cliente actual se introduce mediante la función `leerpantalla` y se escribe en el archivo de datos con la función `escribirarchivo`.

En `main` y `leerpantalla` vemos que los datos se introducen interactivamente, usando las funciones `printf` y `scanf`. Por otro lado, en `main` y `escribirarchivo` los datos se escriben en el archivo mediante la función `fprintf`. La sintaxis que gobierna esta función es la misma que la de `printf`, excepto que se tiene que incluir un puntero a archivo secuencial como argumento adicional. Observe que la cadena de control hace uso de los mismos grupos de caracteres (las mismas características de formato) que la función `printf` descrita en el Capítulo 4.

Cuando se ejecuta el programa, la información para el registro de cada cliente se introducirá interactivamente, como se muestra a continuación para cuatro clientes ficticios. Como siempre, las respuestas del usuario están subrayadas.

```
SISTEMA DE FACTURACION DE CLIENTES - INICIALIZACION

Introduzca la fecha actual (mm/dd/aaaa): 5/24/1998

Nombre (introducir 'FIN' para terminar): Steve Johnson
Calle: 123 Mountainview Drive
Ciudad: Denver, CO
Número de cuenta: 4208
Saldo actual: 247.88
```

```
Nombre (introducir 'FIN' para terminar): Susan Richards
Calle: 4383 Alligator Blvd
Ciudad: Fort Lauderdale, FL
Número de cuenta: 2219
Saldo actual: 135.00

Nombre (introducir 'FIN' para terminar): Martin Peterson
Calle: 1787 Pacific Parkway
Ciudad: San Diego, CA
Número de cuenta: 8452
Saldo actual: 387.42

Nombre (introducir 'FIN' para terminar): Phyllis Smith
Calle: 1000 Great White Way
Ciudad: New York, NY
Número de cuenta: 711
Saldo actual: 260.00

Nombre (introducir 'FIN' para terminar): FIN
```

Después de ejecutar el programa se ha creado el archivo de datos `registro.dat`, que contiene la siguiente información:

```
Steve Johnson
123 Mountainview Drive
Denver, CO
4208
A
247.88
0.00
0.00
5/24/1998

Susan Richards
4383 Alligator Blvd
Fort Lauderdale, FL
2219
A
135.00
0.00
0.00
5/24/1998

Martin Peterson
1787 Pacific Parkway
San Diego, CA
8452
A
387.42
0.00
0.00
5/24/1998
```

```
Phyllis Smith
1000 Great White Way
New York, NY
711
A
260.00
0.00
0.00
5/24/1998

FIN
```

En la siguiente sección veremos un programa que actualiza la información contenida en este archivo.

12.3. PROCESAMIENTO DE UN ARCHIVO

La mayoría de las aplicaciones con archivos requieren que se modifique el archivo cuando se procesa. Por ejemplo, en aplicaciones que involucran el procesamiento de registros de clientes, se puede desear añadir nuevos registros al archivo (ya sea al final del archivo o entre los registros existentes), borrar registros existentes, modificar los contenidos de registros o reordenar los registros. Estos requisitos sugieren varias estrategias computacionales distintas.

Consideremos, por ejemplo, el problema de actualizar los registros dentro de un archivo de datos. Hay varios enfoques a este problema. Quizá el enfoque más obvio es leer cada registro de un archivo, actualizar el registro y después escribir el registro actualizado al mismo archivo. Sin embargo, esta estrategia presenta algunos problemas. En particular, es difícil leer y escribir datos con formato al mismo archivo sin modificar la ordenación de los elementos dentro del archivo. Además, el conjunto original de registros puede volverse inaccesible si algo funciona mal durante la ejecución del programa.

Otro enfoque es trabajar con dos archivos diferentes: un archivo antiguo (la *fuente*) y otro nuevo. Se lee cada registro del archivo antiguo, se actualiza y se escribe al nuevo archivo. Cuando se han actualizado todos los registros, se borra el archivo antiguo o se almacena como copia de seguridad y se renombra el archivo nuevo. Así el nuevo archivo se convierte en la fuente para el siguiente turno de modificaciones.

Históricamente, el origen de este método proviene de los primeros días de la computación, cuando los archivos de datos se almacenaban en cintas magnéticas. Sin embargo, este método se usa todavía, ya que proporciona una serie de archivos fuente antiguos que pueden usarse para generar la historia de un cliente. El archivo fuente más reciente se puede usar también para crear de nuevo el archivo actual si éste se daña o se destruye.

EJEMPLO 12.6. Actualización de un archivo que contiene registros de clientes. El Ejemplo 12.5 presentó un programa para crear un archivo de datos llamado `registro.dat` que contiene registros de clientes. Ahora se presenta un programa para actualizar los registros en el archivo de datos. El programa usará el procedimiento de actualización con dos archivos descrito anteriormente. Se supone que el archivo de datos creado previamente, `registro.dat`, ha sido renombrado como `registro.ant`. Éste será el archivo fuente.

La estrategia general será similar a la descrita en el Ejemplo 12.5. Primero se introduce la fecha actual y se entra en un bucle que lee una serie de registros de clientes de `registro.ant` y escribe los correspondientes registros actualizados en un nuevo archivo de datos llamado `registro.nue`. Cada pasada por el bucle leerá un registro, lo actualizará si es necesario y lo escribirá en `registro.nue`. Observe que todos los registros, actualizados o no, se escribirán en `registro.nue`.

El proceso continuará hasta que se lea el nombre `FIN` del archivo fuente (mayúsculas o minúsculas). Cuando esto suceda, se escribirá `FIN` en el nuevo archivo, indicando una condición de fin de archivo.

A continuación se muestra el programa completo. El programa comienza definiendo la composición de cada registro de cliente, usando las mismas definiciones que en el Ejemplo 12.5. Estas definiciones van seguidas por las definiciones de punteros a archivos secuenciales `antpt` y `nuept`.

Dentro de la función `main`, `registro.ant` es abierto como un archivo de sólo lectura ya existente, y `registro.nue` como un archivo nuevo de sólo escritura. Si `registro.ant` no puede abrirse, se genera un mensaje de error. En otro caso, el programa entra en un bucle `while` que lee los sucesivos registros de clientes de `registro.ant` (en realidad del archivo secuencial `antpt`), actualiza cada registro si es necesario y lo escribe en `registro.nue` (del archivo secuencial `nuept`).

```
/* actualizar un archivo de datos que contiene registros de clientes */

#include <stdio.h>
#include <string.h>

#define NULL    0
#define VERDADERO 1

typedef struct {
     int mes;
     int dia;
     int anio;
} fecha;

typedef struct {
     char nombre[80];
     char calle[80];
     char ciudad[80];
     int num_cuenta;                      /* (entero positivo) */
     int tipo_cuenta;                     /* A (Al día), R (atrasada)
                                             o D (delincuente) */
     float anteriorsaldo;                 /* (cantidad no negativa) */
     float nuevosaldo;                    /* (cantidad no negativa) */
     float pago;                          /* (cantidad no negativa) */
     fecha ultimopago;
} registro;

registro leerarchivo(registro cliente); /* prototipo de función */
registro actualizar(registro cliente);  /* prototipo de función */
void escribirarchivo(registro cliente); /* prototipo de función */

FILE *antpt, *nuept;                    /* punteros a la estructu-
                                           ra predefinida FILE */
int mes, dia, anio;                     /* declaración de variables
                                           globales */
```

```
main()
{
   int indicador = VERDADERO;    /*  declaración de variable local */
   registro cliente;             /*  declaración de variable de estructura */

   /* abrir archivos de datos */
   if ((antpt = fopen("registro.ant", "r")) == NULL)
      printf("\nERROR - No se puede abrir archivo especificado\n");
   else {
       nuept = fopen("registro.nue", "w");

       /* introducir fecha actual */
       printf("SISTEMA DE FACTURACION DE CLIENTES");
       printf("- ACTUALIZACION\n\n");
       printf("Introduzca la fecha actual (mm/dd/aaaa): ");
       scanf("%d/%d/%d", &mes, &dia, &anio);

       /* bucle principal */
       while (indicador) {
           /* leer un nombre del archivo antiguo y escribirlo en el
              nuevo archivo */
           fscanf(antpt, " %[^\n]", cliente.nombre);
           fprintf(nuept, "\n%s\n", cliente.nombre);

           /* comprobación de la condición de parada */
           if (strcmp(cliente.nombre, "FIN") == 0)
              break;

           /* leer el resto de la información del archivo antiguo */
           cliente = leerarchivo(cliente);

           /* solicitar la información actualizada */
           cliente = actualizar(cliente);

           /* escribir la información actualizada en el nuevo
              archivo */
           escribirarchivo(cliente);

   }
   fclose(antpt);
   fclose(nuept);
   } /* fin del else */
}

registro leerarchivo(registro cliente) /* leer el resto de datos
                                          del antiguo archivo */

{
```

```
    fscanf(antpt, " %[^\n]", cliente.calle);
    fscanf(antpt, " %[^\n]", cliente.ciudad);
    fscanf(antpt, " %d", &cliente.num_cuenta);
    fscanf(antpt, " %c", &cliente.tipo_cuenta);
    fscanf(antpt, " %f", &cliente.anteriorsaldo);
    fscanf(antpt, " %f", &cliente.nuevosaldo);
    fscanf(antpt, " %f", &cliente.pago);
    fscanf(antpt, " %d/%d/%d", &cliente.ultimopago.mes,
                               &cliente.ultimopago.dia,
                               &cliente.ultimopago.anio);
    return(cliente);
}
registro actualizar(registro cliente) /* solicitar la nueva in-
                                         formación, actualizar el
                                         registro y mostrar el re-
                                         sumen de datos */
{
    printf("\n\nNombre:     %s", cliente.nombre);
    printf("        Número de cuenta: %d\n", cliente.num_cuenta);
    printf("\nSaldo anterior: %7.2f", cliente.anteriorsaldo);
    printf("      Pago actual: ");
    scanf("%f", &cliente.pago);

    if (cliente.pago > 0) {
        cliente.ultimopago.mes = mes;
        cliente.ultimopago.dia = dia;
        cliente.ultimopago.anio = anio;
       cliente.tipo_cuenta = (cliente.pago < 0.1 * cliente.anteriorsaldo)
                             ? 'R' : 'A';
    }
    else
       cliente.tipo_cuenta = (cliente.anteriorsaldo > 0) ? 'D' : 'A';

    cliente.nuevosaldo = cliente.anteriorsaldo - cliente.pago;
    printf("Nuevo saldo:     %7.2f", cliente.nuevosaldo);

    printf("     Estado de la cuenta: ");
    switch (cliente.tipo_cuenta) {
        case 'A':
                    printf("AL DIA\n");
                    break;
        case 'R':
                    printf("ATRASADA\n");
                    break;
        case 'D':
                    printf("DELINCUENTE\n");
                    break;
        default:
                    printf("ERROR\n");
    }
```

```
   return(cliente);
}

void escribirarchivo(registro cliente) /* escribir la información
                                          actualizada en el nue-
                                          vo archivo */
{
   fprintf(nuept, "%s\n", cliente.calle);
   fprintf(nuept, "%s\n", cliente.ciudad);
   fprintf(nuept, "%d\n", cliente.num_cuenta);
   fprintf(nuept, "%c\n", cliente.tipo_cuenta);
   fprintf(nuept, "%.2f\n", cliente.anteriorsaldo);
   fprintf(nuept, "%.2f\n", cliente.nuevosaldo);
   fprintf(nuept, "%.2f\n", cliente.pago);
   fprintf(nuept, "%d/%d/%d\n", cliente.ultimopago.mes,
                                cliente.ultimopago.dia,
                                cliente.ultimopago.anio);
   return;
}
```

Dentro de `main` se lee cada nombre de cliente del archivo fuente y se escribe a continuación en el archivo nuevo. La información restante de cada registro se lee luego del archivo fuente, se actualiza y se escribe en el nuevo archivo dentro de las funciones `leerarchivo`, `actualizar` y `escribirarchivo`, respectivamente. Este proceso continúa hasta que se encuentra un registro que tiene como nombre de cliente `FIN`. A continuación se cierran ambos archivos de datos y termina la ejecución.

La función `leerarchivo` lee la información adicional para cada registro de cliente del archivo fuente. Los distintos datos se representan como miembros de la variable de estructura `cliente`. Esta variable de estructura se pasa como argumento a la función. Se usa la función de biblioteca `fscanf` para leer cada dato, empleando una sintaxis esencialmente idéntica a la usada en la función `scanf`, como se describió en el Capítulo 4. Sin embargo, en `fscanf` hay que incluir el puntero a archivo secuencial `antpt` como argumento adicional dentro de cada llamada a función. Una vez que se ha leído toda la información del archivo fuente, el registro del cliente es devuelto a `main`.

La función `actualizar` es similar, aunque es necesario que se introduzca desde el teclado un valor para `cliente.pago`. A continuación se asigna información adicional a `cliente.ultimopago`, `cliente.tipo_cuenta` y `cliente.nuevosaldo`. Los valores asignados dependen del valor dado a `cliente.pago`. El registro actualizado es devuelto a `main`.

La función restante, `escribirarchivo`, acepta cada registro de cliente como un argumento y lo escribe en el nuevo archivo de datos. Dentro de `escribirarchivo` la función de biblioteca `fprintf` se usa para transferir la información al nuevo archivo de datos usando los mismos procedimientos mostrados en el Ejemplo 12.5.

Cuando se ejecuta el programa, se muestran para cada cliente el nombre, número de cuenta y saldo anterior. A continuación se pide al usuario un valor para el pago actual. Cuando se ha introducido este valor, se muestran el nuevo saldo y el estado actual de la cuenta del cliente.

A continuación se muestra una típica sesión interactiva basada en el archivo creado en el Ejemplo 12.5. Las respuestas del usuario están subrayadas, como siempre.

```
SISTEMA DE FACTURACION DE CLIENTES - ACTUALIZACION

Introduzca la fecha actual (mm/dd/aaaa): 12/29/1998

Nombre:    Steve Johnson       Número de cuenta: 4208

Saldo anterior: 247.88  Pago actual: 25.00
Nuevo saldo:    222.88  Estado de la cuenta: AL DIA


Nombre:    Susan Richards      Número de cuenta: 2219

Saldo anterior: 135.00  Pago actual: 135.00
Nuevo saldo:      0.00  Estado de la cuenta: AL DIA


Nombre:    Martin Peterson      Número de cuenta: 8452

Saldo anterior: 387.42  Pago actual: 35.00
Nuevo saldo:    352.42  Estado de la cuenta: ATRASADA


Nombre:    Phyllis Smith       Número de cuenta: 711

Saldo anterior: 260.00  Pago actual: 0
Nuevo saldo:    260.00  Estado de la cuenta: DELINCUENTE
```

Una vez que se han procesado todos los registros de clientes, se habrá creado el nuevo archivo de datos `registro.nue`, que contiene la siguiente información.

```
Steve Johnson
123 Mountainview Drive
Denver, CO
4208
A
247.88
222.88
25.00
12/29/1998

Susan Richards
4383 Alligator Blvd
Fort Lauderdale, FL
2219
A
135.00
0.00
135.00
12/29/1998
```

```
Martin Peterson
1787 Pacific Parkway
San Diego, CA
8452
R
387.42
352.42
35.00
12/29/1998

Phyllis Smith
1000 Great White Way
New York, NY
711
D
260.00
260.00
0.00
5/24/1998

FIN
```

Observe que el archivo antiguo, `registro.ant`, aún está disponible en su forma original; por tanto, puede ser almacenado como archivo histórico. Antes de volver a ejecutar este programa, el nuevo archivo de datos ha de renombrarse como `registro.ant`. (Generalmente esto se hace a nivel de sistema operativo, antes de entrar en el programa de actualización.)

12.4. ARCHIVOS SIN FORMATO

Algunas aplicaciones involucran el uso de archivos para almacenar bloques de datos, donde cada bloque consiste en un número fijo de bytes contiguos. Cada bloque representará generalmente una estructura de datos compleja, como una estructura o un array. Por ejemplo, un archivo de datos puede constar de varias estructuras que tengan la misma composición, o puede contener múltiples arrays del mismo tipo y tamaño. Para estas aplicaciones sería deseable leer o escribir el bloque entero del archivo de datos en vez de leer o escribir separadamente las componentes individuales (los miembros de la estructura o elementos del array) de cada bloque.

Las funciones de biblioteca `fread` y `fwrite` deben usarse en estas situaciones. A estas funciones se las conoce frecuentemente como funciones de lectura y escritura *sin formato*. Igualmente se hace referencia a estos archivos de datos como archivos de datos sin formato.

Cada una de estas funciones necesita cuatro argumentos: un puntero al bloque de datos, el tamaño del bloque de datos, el número de bloques a transferir y el puntero a un archivo secuencial. Así, una función `fwrite` típica podría escribirse como

```
fwrite(&cliente, sizeof(registro), 1, fpt);
```

donde `cliente` es una variable de estructura de tipo `registro` y `fpt` es un puntero a archivo secuencial asociado a un archivo de datos abierto para salida.

EJEMPLO 12.7. Creación de un archivo de datos sin formato que contiene registros de clientes. Consideremos una variación del programa presentado en el Ejemplo 12.5 para crear un archivo de datos que contiene registros de clientes. Sin embargo, ahora se escribe cada registro de cliente en el archivo `datos.bin` como un bloque de información simple, sin formato. Esto está en contraste con el programa anterior, donde se escribieron los elementos de cada registro (los miembros de la estructura individual) como datos separados y con formato.

A continuación se muestra el programa completo.

```
/* crear un archivo de datos sin formato que contiene registros de
   clientes */

#include <stdio.h>
#include <string.h>

#define VERDADERO 1

typedef struct {
     int mes;
     int dia;
     int anio;
} fecha;

typedef struct {
     char nombre[80];
     char calle[80];
     char ciudad[80];
     int num_cuenta;                      /* (entero positivo) */
     int tipo_cuenta;                     /* A (Al día), R (atrasa-
                                             da) o D (delincuente) */
     float anteriorsaldo;                 /* (cantidad no negativa) */
     float nuevosaldo;                    /* (cantidad no negativa) */
     float pago;                          /* (cantidad no negativa) */
     fecha ultimopago;
} registro;

registro leerpantalla(registro cliente); /* prototipo de función */

FILE *fpt;        /* puntero a la estructura predefinida FILE */

main()
{
   int indicador = VERDADERO;         /* declaración de variable */
   registro cliente;       /* declaración de variable de estructura */

   /* abrir un archivo nuevo solo para escritura */
   fpt = fopen("datos.bin", "w");

   /* introducir datos y asignar valores iniciales */
   printf("SISTEMA DE FACTURACION DE CLIENTES - INICIALIZACION\n\n");
   printf("Introduzca la fecha actual (mm/dd/aaaa): ");
```

```
    scanf("%d/%d/%d",  &cliente.ultimopago.mes,
                       &cliente.ultimopago.dia,
                       &cliente.ultimopago.anio);
    cliente.nuevosaldo = 0;
    cliente.pago = 0;
    cliente.tipo_cuenta = 'A';

    /* bucle principal */
    while (indicador) {
        /* introducir el nombre del cliente */
        printf("\nNombre (introducir \'FIN\' para terminar): ");
        scanf(" %[^\n]", cliente.nombre);

        /* comprobación de la condición de parada */
        if (strcmp(cliente.nombre, "FIN") == 0)
            break;

        /* introducir el resto de los datos y escribirlo en el archivo */
        cliente = leerpantalla(cliente);
        fwrite(&cliente, sizeof(registro), 1, fpt);

        /* borrar cadenas de caracteres */
        strset(cliente.nombre, ' ');
        strset(cliente.calle, ' ');
        strset(cliente.ciudad, ' ');
    }

    fclose(fpt);
}

registro leerpantalla(registro cliente)  /* leer el resto de datos */

{
    printf("Calle: ");
    scanf(" %[^\n]", cliente.calle);
    printf("Ciudad: ");
    scanf(" %[^\n]", cliente.ciudad);
    printf("Número de cuenta: ");
    scanf("%d", &cliente.num_cuenta);
    printf("Saldo actual: ");
    scanf("%f", &cliente.anteriorsaldo);
    return(cliente);
}
```

Comparando este programa con el mostrado en el Ejemplo 12.5, se observa que ambos programas son muy similares. En `main`, el programa actual lee cada nombre de cliente y comprueba una condición de finalización (`FIN`), pero no escribe el nombre del cliente en el archivo de datos, como en el programa anterior. Por el contrario, si no se indica una condición de finalización, se lee interactivamente el resto del registro del cliente y se escribe en el archivo de datos con la instrucción `fwrite`

```
fwrite(&cliente, sizeof(registro), 1, fpt);
```

Observe que el archivo creado por este programa se llama `datos.bin`, como se indicó en el primer argumento en la llamada a la función `fopen`.

La función definida por el programador `escribirarchivo` mostrada en el Ejemplo 12.5 no es necesaria en este programa, pues la función de biblioteca `fwrite` toma su lugar. Por otro lado, ambos programas usan la función definida por el programador `leerpantalla`, que hace que la información dada para un registro de cliente se introduzca en la computadora interactivamente.

Después de que cada registro se ha escrito en el archivo de datos, se borran los miembros que son cadenas de caracteres `cliente.nombre`, `cliente.calle` y `cliente.ciudad` (sustituidos con espacios en blanco), de forma que nada de la información previa se incluirá en cada nuevo registro. La función de biblioteca `strset` se usa con este propósito. Así, la declaración

```
strset(cliente.nombre, ' ');
```

reemplaza con espacios en blanco el contenido de `cliente.nombre`, como indica `' '`. Observe que se ha incluido el archivo de cabecera `string.h` como soporte a la función `strset`.

La ejecución de este programa produce el mismo diálogo interactivo que el mostrado en el Ejemplo 12.5. De esta forma durante la ejecución del programa el usuario no puede decir si el archivo está siendo creado con o sin formato. Una vez que el nuevo archivo `datos.bin` haya sido creado, sus contenidos no serán legibles a menos que se lea el archivo con un programa especialmente escrito para ello. Este programa se muestra en el siguiente ejemplo.

Una vez que se ha creado un archivo de datos sin formato, surge la pregunta de cómo detectar una condición de fin de archivo. La función de biblioteca `feof` sirve para este propósito. (Realmente `feof` indicará una condición de fin de archivo para *cualquier* archivo de datos secuencial, no sólo para uno sin formato.) Esta función devuelve un valor distinto de cero (`VERDADERO`) si se detecta una condición de fin de archivo y un valor cero (`FALSO`) si *no* se detecta. Por tanto, un programa que lee un archivo de datos sin formato puede utilizar un bucle que lea los sucesivos registros hasta que el valor devuelto por `feof` sea `no VERDADERO`.

EJEMPLO 12.8. Actualización de un archivo de datos sin formato que contiene registros de clientes. Se considera ahora otro programa para leer y actualizar el archivo sin formato creado en el Ejemplo 12.7. Se volverá a hacer uso de un procedimiento de actualización con dos archivos, como en el Ejemplo 12.6. Sin embargo, ahora los archivos se llamarán `datos.ant` y `datos.nue`. Por tanto, el archivo creado en el ejemplo anterior, llamado `datos.bin`, se renombrará como `datos.ant` antes de ejecutar el programa actual.

La lógica general del programa es similar a la presentada en el Ejemplo 12.6. Esto es, se lee un registro de `datos.ant`, se actualiza interactivamente y se escribe en `datos.nue`. Este proceso continúa hasta que se detecte la condición de fin de archivo en la operación `fread` más reciente. Observe la forma en la cual se construye la comprobación de fin de archivo en el bucle `while`, esto es, `while (!feof(antpt))`.

Sin embargo, este programa hará uso de las funciones de biblioteca `fread` y `fwrite` para leer registros de clientes sin formato de `datos.ant` y para escribir los registros actualizados en `datos.nue`. Por tanto, el programa actual no hace uso de las funciones definidas por el programador `leerarchivo` y `escribirarchivo`, como en el Ejemplo 12.6.

A continuación se muestra el programa completo en C.

```
/* actualizar un archivo de datos sin formato que contiene regis-
   tros de clientes */
```

```
#include <stdio.h>

#define NULL 0

typedef struct {
     int mes;
     int dia;
     int anio;
} fecha;

typedef struct {
     char nombre[80];
     char calle[80];
     char ciudad[80];
     int num_cuenta;                    /* (entero positivo) */
     int tipo_cuenta;                   /* A (Al día), R (atrasa-
                                           da) o D (delincuente) */
     float anteriorsaldo;               /* (cantidad no negativa) */
     float nuevosaldo;                  /* (cantidad no negativa) */
     float pago;                        /* (cantidad no negativa) */
     fecha ultimopago;
} registro;

registro actualizar(registro cliente);/* prototipo de función */
FILE *antpt, *nuept;                  /* punteros a la estructu-
                                         ra predefinida FILE */
int mes, dia, anio;         /* declaración de variables globales */

main()

{
  registro cliente;      /* declaración de variable de estructura */

  /* abrir archivos de datos */
  if ((antpt = fopen("datos.ant", "r")) == NULL)
     printf("\nERROR - No se puede abrir el archivo especificado\n");
  else {
      nuept = fopen("datos.nue", "w");

      /* introducir fecha actual */
     printf("SISTEMA DE FACTURACION DE CLIENTES - ACTUALIZACION\n\n");
      printf("Introduzca la fecha actual (mm/dd/aaaa): ");
      scanf("%d/%d/%d", &mes, &dia, &anio);

      /* leer el primer registro del archivo antiguo de datos */
      fread(&cliente, sizeof(registro), 1, antpt);

      /* bucle principal (continuar hasta que se detecte
                            fin-de-archivo) */
```

```
        while (!feof(antpt)) {
            /* solicitar la información actualizada */
            cliente = actualizar(cliente);

            /* escribir la información actualizada en el nuevo archivo */
            fwrite(&cliente, sizeof(registro), 1, nuept);

            /* leer el siguiente registro del archivo antiguo de datos */
            fread(&cliente, sizeof(registro), 1, antpt);
        }

        fclose(antpt);
        fclose(nuept);
    } /* fin del else */
}

registro actualizar(registro cliente) /* solicitar la nueva in-
                                         formación, actualizar el
                                         registro y mostrar el re-
                                         sumen de datos */
{
    printf("\n\nNombre:   %s", cliente.nombre);
    printf("      Número de cuenta: %d\n", cliente.num_cuenta);
    printf("\nSaldo anterior: %7.2f", cliente.anteriorsaldo);
    printf("      Pago actual: ");
    scanf("%f", &cliente.pago);

    if (cliente.pago > 0) {
        cliente.ultimopago.mes = mes;
        cliente.ultimopago.dia = dia;
        cliente.ultimopago.anio = anio;
        cliente.tipo_cuenta = (cliente.pago < 0.1 * cliente.anteriorsaldo)
                              ? 'R' : 'A';
    }
    else
        cliente.tipo_cuenta = (cliente.anteriorsaldo > 0) ? 'D' : 'A';

    cliente.nuevosaldo = cliente.anteriorsaldo - cliente.pago;
    printf("Nuevo saldo:      %7.2f", cliente.nuevosaldo);

    printf("      Estado de la cuenta: ");
    switch (cliente.tipo_cuenta) {
        case 'A':
                    printf("AL DIA\n");
                    break;
        case 'R':
                    printf("ATRASADA\n");
                    break;
        case 'D':
                    printf("DELINCUENTE\n");
                    break;
```

```
            default:
                        printf("ERROR\n");
        }
        return(cliente);
    }
```

Los resultados de la ejecución de este programa son similares a los del Ejemplo 12.6.

No se continuará más allá empleando archivos de datos en este libro. Recordar, sin embargo, que la mayoría de las versiones de C contienen muchas funciones de biblioteca para realizar operaciones orientadas a archivos. Algunas de estas funciones se usan con los dispositivos estándar de entrada/salida (lectura del teclado y escritura en pantalla), otras se usan para archivos de datos secuenciales y otras están disponibles para archivos de datos orientados al sistema. En este capítulo tan sólo hemos arañado la superficie de este importante tema. Se anima al lector a descubrir qué funciones relacionadas con archivos están disponibles en su versión particular del lenguaje.

CUESTIONES DE REPASO

12.1. ¿Cuál es la ventaja fundamental de usar archivos?

12.2. Describir las distintas formas en que pueden clasificarse los archivos en C.

12.3. ¿Cuál es el propósito del área de buffer cuando se trabaja con un archivo secuencial de datos? ¿Cómo se define el área de buffer?

12.4. Cuando se define el área de buffer para usarla con un archivo secuencial, ¿qué representa el símbolo FILE? ¿Dónde se define `FILE`?

12.5. ¿Qué es un puntero a un archivo secuencial? ¿Cuál es la relación entre un puntero a un archivo secuencial y un área de buffer?

12.6. ¿Qué se entiende por abrir un archivo? ¿Cómo se realiza?

12.7. Relacionar las reglas que gobiernan el uso de la función `fopen`. Describir la información que devuelve esta función.

12.8. Relacionar los diferentes tipos de archivos que pueden ser especificados mediante la función `fopen`.

12.9. ¿Cuál es el propósito de la función `fclose`? ¿Debe aparecer una llamada a esta función dentro de un programa que usa un archivo?

12.10. Describir una construcción normalmente usada en programación que presente un mensaje de error asociado a la llamada a la función `fopen`.

12.11. Describir dos métodos distintos para crear un archivo de datos secuencial. ¿Pueden usarse los dos métodos en archivos sin formato?

12.12. Describir el procedimiento general para crear un archivo de datos secuencial mediante un programa especial escrito en C. ¿Qué funciones de biblioteca orientadas a archivos deben usarse dentro del programa?

12.13. ¿Cómo puede visualizarse un archivo secuencial una vez que ha sido creado? ¿Se puede aplicar la respuesta a archivos sin formato?

12.14. Describir el procedimiento general para leer archivos de datos secuenciales usando un programa escrito en C. ¿Qué funciones de biblioteca orientadas a archivos deben usarse dentro del programa? Comparar la respuesta con la del Problema 12.12.

12.15. Describir dos enfoques distintos para actualizar un archivo de datos. ¿Qué enfoque es mejor y por qué?

12.16. Contrastar el uso de las funciones `fscanf` y `fprintf` con el uso de `scanf` y `printf` descritas en el Capítulo 4. ¿En qué se diferencian las reglas gramaticales?

12.17. ¿Para qué tipo de aplicaciones son útiles los archivos sin formato?

12.18. Contrastar el uso de las funciones `fread` y `fwrite` con la utilización de `fscanf` y `fprintf`. ¿En qué se diferencian las reglas gramaticales? ¿Para qué tipo de aplicaciones es adecuado cada grupo de funciones?

12.19. ¿Cuál es el propósito de la función de biblioteca `strset`? ¿Por qué debería usarse la función de biblioteca `strset` en un programa que crea un archivo sin formato?

12.20. ¿Cuál es el propósito de la función de biblioteca `feof`? ¿Cómo debe utilizarse la función de biblioteca `feof` dentro de un programa que actualiza un archivo de datos sin formato?

PROBLEMAS

12.21. Asociar el puntero a archivo secuencia `puntr` con un nuevo archivo secuencial llamado `estudian.dat`. Abrir el archivo sólo para escritura.

12.22. Asociar el puntero a archivo secuencial `puntr` con un archivo secuencial previamente existente llamado `estudian.dat`. Abrir el archivo de modo que se pueda leer y escribir en él. Mostrar cómo se debe cerrar el archivo al final del programa.

12.23. Asociar el puntero a archivo secuencial `puntr` con un archivo secuencial nuevo llamado `muestra.dat`. Abrir el archivo de modo que se pueda leer y escribir en él. Mostrar cómo se puede cerrar el archivo al final del programa.

12.24. Asociar el puntero a archivo secuencial `puntr` con un archivo secuencial previamente existente llamado `muestra.dat`. Abrir el archivo de modo que se pueda leer y escribir en él. Mostrar cómo puede cerrarse el archivo al final del programa.

12.25. Repetir el Problema 12.24, añadiendo la posibilidad de generar un mensaje de error en el caso de que el archivo de datos no pueda ser abierto (por ejemplo, si el archivo no está presente).

12.26. A continuación se muestra el esquema de la estructura de un programa en C.

```
#include <stdio.h>

main()
{
   FILE *fpt;

   int a;
   float b;
   char c;

   fpt = fopen("muestra.dat", "w");
   . . . . .

   fclose(fpt);
}
```

Introducir los valores para a, b y c desde el teclado, en respuesta a las peticiones generadas por el programa. Después escribir los valores en el archivo. Formatear los valores en coma flotante de modo que se escriban en el archivo con no más de dos decimales.

12.27. A continuación se muestra el esquema de la estructura de un programa en C.

```
#include <stdio.h>
main()
{
  FILE *fpt;

  int a;
  float b;
  char c;

  fpt = fopen("muestra.dat", "r");

  . . . . .

  fclose(fpt);
}
```

Leer los valores de a, b y c del archivo y mostrarlos en la pantalla.

12.28. A continuación se muestra el esquema de la estructura de un programa en C.

```
#include <stdio.h>
main()
{
  FILE *pt1, *pt2;

  int a;
  float b;
  char c;

  pt1 = fopen("muestra.ant", "r");
  pt2 = fopen("muestra.nue", "w");

  . . . . .

  fclose(pt1);
  fclose(pt2);
}
```

a) Leer los valores de a, b y c del archivo muestra.ant.
b) Mostrar cada valor en la pantalla e introducir un valor actualizado.
c) Escribir los nuevos valores en el archivo muestra.nue. Formatear los valores en coma flotante de modo que se escriban en el archivo con no más de dos decimales.

12.29. A continuación se muestra el esquema de la estructura de un programa en C.

```
#include <stdio.h>
main()
{
    FILE *pt1, *pt2;

    char nombre[20];

    pt1 = fopen("muestra.ant", "r");
    pt2 = fopen("muestra.nue", "w");
    .....
    fclose(pt1);
    fclose(pt2);
  }
```

a) Leer la cadena de caracteres representada por `nombre` del archivo `muestra.ant`.
b) Mostrarla en la pantalla.
c) Introducir una nueva cadena.
d) Escribir la nueva cadena al archivo `muestra.nue`.

12.30. A continuación se muestra el esquema de la estructura de un programa en C.

```
#include <stdio.h>
main()
{
  FILE *pt1, *pt2;

  struct {
      int a;
      float b;
      char c;
      char nombre[20];
  } valores;

  pt1 = fopen("datos.ant", "r");
  pt2 = fopen("datos.nue", "w+");

  . . . . .

  fclose(pt1);
  fclose(pt2);
}
```

a) Leer el valor de `valores.nombre` del archivo con formato `datos.ant` y mostrarlo en la pantalla.
b) Introducir los valores para `valores.a`, `valores.b` y `valores.c` desde el teclado, como respuesta a las peticiones del programa.
c) Escribir los valores de `valores.nombre`, `valores.a`, `valores.b` y `valores.c` en el archivo con formato `datos.nue`.

12.31. Repetir el Problema 12.30 tratando los dos archivos de datos como archivos sin formato. (Leer un registro completo de `datos.ant` y escribir el registro actualizado en `datos.nue`.)

PROBLEMAS DE PROGRAMACIÓN

12.32. Modificar el programa dado en el Ejemplo 12.3 (lectura de una línea de texto en minúsculas y escribirla en mayúsculas en un archivo) de modo que cada carácter introducido por teclado sea comprobado para ver si está en mayúsculas o minúsculas, y se escriba en el archivo de modo opuesto. (Por tanto, las minúsculas se convierten a mayúsculas y las mayúsculas a minúsculas.) Usar las funciones de biblioteca `isupper` o `islower` para comprobar si cada carácter es minúscula o mayúscula, y las funciones `toupper` y `tolower` para realizar las conversiones.

12.33. Modificar los programas dados en los Ejemplos 12.3 y 12.4 de modo que se puedan procesar varias líneas de texto. Como condición de finalización, verificar si está presente la cadena `FIN` (en minúsculas o mayúsculas) en los tres primeros caracteres dentro de cada línea.

12.34. Modificar el programa dado en el Ejemplo 12.6 (actualización de un archivo de registros de clientes) de modo que use sólo un archivo; esto es, cada registro de cliente actualizado reemplaza al original. Utilizar la función de biblioteca `ftell` para determinar la posición actual en el archivo y la función `fseek` para cambiar la posición actual en el archivo según se necesite. Asegurarse de abrir el archivo en el modo adecuado.

12.35. Ampliar el programa dado en el Ejemplo 12.6 de modo que se puedan añadir nuevos registros, borrar registros antiguos y modificar registros existentes. Asegurarse de mantener los registros en orden alfabético. Permitir que el usuario escoja qué opción se ejecutará antes de procesar el registro.

12.36. Modificar el programa dado en el Ejemplo 12.8 (actualizar un archivo sin formato que contiene registros de clientes) de modo que use sólo un archivo; esto es, cada registro de cliente actualiza-do reemplaza al original. Utilizar la función de biblioteca `ftell` para determinar la posición actual en el archivo y la función `fseek` para cambiar la posición en el archivo según se necesite. Asegurarse de abrir el archivo en el modo adecuado.

12.37. Escribir un programa que lea registros sucesivos del archivo nuevo creado en el Ejemplo 12.8 y muestre cada registro en la pantalla adecuadamente formateado.

12.38. Ampliar el programa descrito en el Problema 12.36 de modo que se puedan añadir nuevos registros, borrar registros antiguos y modificar registros existentes. Asegurarse de mantener los registros en orden alfabético. Permitir que el usuario escoja qué opción se ejecutará antes de procesar el registro.

12.39. Escribir un programa interactivo en C que codifique y decodifique varias líneas de texto usando el procedimiento de codificación/decodificación descrito en el Problema 9.49. Almacenar el texto codificado en un archivo de modo que pueda ser recuperado y decodificado en cualquier momento. El programa debe incluir las siguientes características:

a) Introducir el texto desde teclado, codificarlo y almacenar el texto codificado en un archivo de datos.
b) Recuperar el texto codificado y mostrarlo en su forma codificada.
c) Recuperar el texto codificado, decodificarlo y mostrarlo en su forma decodificada.
d) Finalizar la ejecución.

Probar el programa utilizando varias líneas de texto de su elección.

12.40. Ampliar el programa dado en el Ejemplo 12.39 de modo que se puedan generar enteros aleatorios, donde cada entero sucesivo se utiliza para decodificar cada línea consecutiva. Así, el primer entero aleatorio será usado para codificar la primera línea de texto, el segundo para codificar la segunda línea, y así sucesivamente. Incluir la posibilidad de reproducir la secuencia de enteros aleatorios, de modo que los mismos enteros aleatorios puedan ser utilizados para decodificar el texto. Comprobar el programa introduciendo varias líneas de su elección.

12.41. Modificar el simulador del juego «Craps» dado en el Ejemplo 7.11 de modo que simule un número especificado de juegos y grabe el resultado en archivo. Al final de la simulación, leer el archivo de datos para calcular el porcentaje de victorias y derrotas de cada jugador.

Comprobar el programa simulando 500 juegos consecutivos. Basándose en estos resultados, calcular las posibilidades de ganar.

12.42. Modificar el generador de «pig latin» presentado en el Ejemplo 9.14 de modo que puedan introducirse varias líneas de texto desde el teclado. Grabar el texto completo en un archivo y el correspondiente «pig latin» en otro.
Incluir dentro del programa la generación de un menú que permita al usuario seleccionar cualquiera de las siguientes características:

a) Introducir texto nuevo, convertirlo a «pig latin» y grabarlo. (Grabar tanto el texto original como el «pig latin», según se describió anteriormente.)
b) Leer un texto previamente introducido en un archivo y mostrarlo.
c) Leer el texto «pig latin» equivalente del introducido previamente y mostrarlo.
d) Finalizar la ejecución.

Comprobar el programa utilizando varias línea de texto arbitrarias.

12.43. Escribir un programa completo en C que genere un archivo que contiene los datos de las notas de los exámenes de los estudiantes presentados en el Problema 6.69(*k*). Cada componente del archi-vo será una estructura que contenga el nombre y las notas de examen para un estudiante. Ejecutar el programa creando un archivo para usarlo en el siguiente problema.

12.44. Escribir un programa en C orientado a archivo que procesará las notas de los exámenes de los estudiantes dadas en el Problema 6.69(*k*). Leer los datos del archivo creado en el problema anterior. Crear a continuación un informe que contenga el nombre, las notas de los exámenes y la media de ellas para cada estudiante.

12.45. Ampliar el programa escrito para el Problema 12.44 de modo que se determine una media general de la clase, seguida por la desviación media de cada estudiante respecto de la media general. Escribir la salida en un archivo nuevo. Después presentar la salida como un informe bien rotulado.

12.46. Escribir un programa en C interactivo y orientado a archivo que mantenga una lista de nombres, direcciones y números de teléfono en orden alfabético (por apellidos). Procesar la información asociada con cada nombre como un registro separado. Representar cada registro como una estructura.

Incluir un menú que permitirá al usuario seleccionar cualquiera de las siguientes caracte-rís-ticas:

a) Añadir un nuevo registro.
b) Borrar un registro.
c) Modificar un registro existente.
d) Recuperar y mostrar un registro completo para un nombre dado.
e) Generar una lista completa de todos los nombres, direcciones y números de teléfono.
f) Finalizar la ejecución.

Asegurarse de reordenar los registros cada vez que se añada un registro nuevo o si se borra un registro, de modo que todos los registros se mantengan siempre en orden alfabético. Utilizar una lista lineal enlazada, como se describe en el Ejemplo 11.32.

12.47. Escribir un programa en C que genere un archivo que contiene la lista de países y sus correspondientes capitales, dadas en el Problema 9.46. Colocar el nombre de cada país y su capital correspondiente en una estructura. Tratar cada estructura como un registro separado. Ejecutar el programa creando un archivo para usarlo en el siguiente problema.

12.48. Escribir un programa interactivo en C, guiado por menús, que acceda al archivo generado en el problema anterior y permita ejecutar una de las siguientes operaciones:

a) Determinar la capital de un país especificado.
b) Determinar el país cuya capital se especifica.
c) Terminar la ejecución.

12.49. Ampliar el programa escrito para el Problema 12.48 para incluir las siguientes características adicionales:

a) Añadir un nuevo registro (un país nuevo y su correspondiente capital).
b) Borrar un registro.
c) Generar un listado de todos los países y sus capitales correspondientes.

12.50. Escribir un programa en C que pueda ser usado como un editor de texto orientado a líneas. Este programa debe tener las siguientes posibilidades:

a) Introducir varias líneas de texto y almacenarlas en un archivo.
b) Listar el archivo.
c) Recuperar y mostrar una línea particular, determinada por su número de línea.
d) Insertar *n* líneas.
e) Borrar *n* líneas.
f) Grabar el nuevo texto editado y finalizar la ejecución.

Realizar cada una de estas tareas respondiendo a una orden compuesta por una letra precedida por el signo del dólar (`$`). La orden buscar (recuperar) debe ir seguida por un entero sin signo para indicar la línea que debe recuperarse. Las órdenes de inserción y borrado pueden ir seguidas por un entero sin signo para indicar la línea que debe recuperarse. Las órdenes de inserción y borrado pueden ir seguidas por un entero sin signo opcional, si se quiere insertar o borrar varias líneas consecutivas.

Cada orden debe aparecer en una línea, proporcionando así un método de distinguir las órdenes de las líneas de texto. (Una línea de orden empezará siempre con el signo del dólar, seguido por una orden de una letra, un entero sin signo opcional y un símbolo de nueva línea.)

Se recomiendan las siguientes órdenes:

`$E` -- introducir texto.
`$L` -- listar el bloque de texto completo.
`$F`*k* -- buscar (recuperar) la línea número *k*.
`$I`*n* -- insertar *n* líneas después de la línea número *k*.
`$D`*n* -- borrar *n* líneas después de la línea número *k*.
`$S` -- grabar el bloque de texto editado y terminar la ejecución.

12.51. Ampliar el programa descrito en el Problema 11.67 de modo que la información del equipo se mantenga en un archivo en vez de en un array. Cada componente del archivo debe ser una estructura que contenga los datos de un equipo. Incluir las siguientes operaciones:

a) Introducir nuevos registros (añadir nuevos equipos).
b) Actualizar registros existentes.
c) Borrar registros (quitar equipos).
d) Generar un informe resumen para todos los equipos de la liga.

CAPÍTULO 13

Programación a bajo nivel

De la materia presentada en los primeros doce capítulos de este libro debería haber quedado claro que C es un lenguaje de programación con todas las características de alto nivel. Sin embargo, C también posee ciertas características de «bajo nivel» que permiten al programador realizar ciertas operaciones que normalmente sólo son posibles en lenguaje ensamblador o en lenguaje máquina. Por ejemplo, es posible almacenar los valores de ciertas variables en los registros de la unidad central de procesamiento. Esto normalmente acelerará cualquier cálculo asociado con esos valores.

Además C permite la manipulación de bits individuales dentro de una palabra. Así, los bits pueden ser desplazados hacia la izquierda o hacia la derecha, *invertidos* (unos y ceros invertidos) o *enmascarados* (extraídos selectivamente). Las aplicaciones que requieren tales operaciones son familiares a los programadores en lenguaje ensamblador. C también permite organizar los bits dentro de una palabra en grupos individuales. Esto permite empaquetar varios datos en una sola palabra.

Este capítulo muestra cómo realizar operaciones a bajo nivel en C. Los lectores que no tengan experiencia en este área podrían desear pasar por alto parte de esta materia, en particular la sección 13.2.

13.1. VARIABLES REGISTRO

En el Capítulo 8 se mencionó que existen cuatro especificaciones de tipo de almacenamiento en C y se examinaron tres de ellas en detalle (automática, externa y estática). Ahora se presta atención al último de ellos, el tipo de almacenamiento *registro*.

Los registros son áreas especiales de almacenamiento dentro de la unidad central de procesamiento de la computadora. Las operaciones aritméticas y lógicas reales se realizan dentro de estos registros. Normalmente estas operaciones se efectúan transfiriendo información desde la memoria de la computadora hasta estos registros, realizando las operaciones indicadas y transfiriendo de nuevo los resultados a la memoria de la computadora. Este procedimiento general se repite muchas veces durante el curso de la ejecución de un programa.

Para algunos programas el tiempo de ejecución se puede reducir considerablemente si ciertos valores pueden almacenarse dentro de los registros en vez de en la memoria de la computadora. Además, tales programas pueden ser algo menores en tamaño (pueden requerir menos instrucciones), ya que se necesitan menos transferencias. Sin embargo, normalmente no se reducirá el tamaño espectacularmente y será menos significativo que el ahorro en tiempo de ejecución.

En C, los valores de las *variables registro* se almacenan dentro de los registros de la unidad central de procesamiento. A una variable se le puede asignar este tipo de almacenamiento simple-

mente precediendo la declaración de tipo con la palabra reservada `register`. Pero sólo puede haber unas pocas variables registro (normalmente, dos o tres) dentro de cualquier función. El número exacto depende de la computadora particular y del compilador específico de C. Normalmente sólo las variables enteras son asignadas al tipo de almacenamiento `register` (más sobre esto posteriormente en esta sección).

Los tipos de almacenamiento `register` y `automatic` están muy relacionados. En particular, su visibilidad (su ámbito) es la misma. Por tanto, las variables `register`, al igual que las variables `automatic`, son locales a la función en la que han sido declaradas. Además, las reglas que gobiernan el uso de las variables `register` son las mismas que las de las variables `automatic` (ver sección 8.2), excepto que el operador dirección (`&`) no se puede aplicar a las variables registro.

Estas semejanzas entre las variables `register` y `automatic` no son casuales, porque el tipo de almacenamiento `register` sólo puede asignarse a variables que de otro modo tendrían el tipo de almacenamiento `automatic`. Además, el declarar ciertas variables como `register` no garantiza que sean tratadas realmente como variables de tipo `register`. La declaración será válida sólo si el espacio requerido de registro está disponible. Si una declaración `register` no se puede tener en cuenta, las variables se tratarán como si tuvieran el tipo de almacenamiento `automatic`.

EJEMPLO 13.1. Un programa en C contiene la declaración de variable

```
register int a, b, c;
```

Esta declaración especifica que las variables `a`, `b` y `c` serán variables enteras con tipo de almacenamiento `register`. Por tanto, los valores de `a`, `b` y `c` se almacenarán dentro de los registros de la unidad central de procesamiento de la computadora en vez de en la memoria, siempre que exista espacio disponible en los registros.

Si no existe espacio disponible en los registros, las variables se tratarán como enteras con tipo de almacenamiento `automatic`. Esto es equivalente a la declaración

```
auto int a, b, c;
```

o simplemente

```
int a, b, c;
```

como se explicó en la sección 8.2.

Desafortunadamente, no hay otra forma de determinar si la declaración `register` se ha tenido en cuenta que ejecutar el programa con y sin la declaración y comparar los resultados. Un programa que hace uso de variables registro debe ejecutarse más rápido que el programa correspondiente sin las variables registro. Puede ser también de tamaño algo menor.

EJEMPLO 13.2. **Generación de los números de Fibonacci.** El programa presentado a continuación es una variación del mostrado en el Ejemplo 8.7 para generar una serie de números de Fibonacci.

```
/* calcular 10 000 000 veces los primeros 23 números de Fibonacci,
   para ilustrar el uso de variables registro */

#include <stdio.h>
#include <time.h>

main()
{
   time_t prin, fin;          /* tiempos de comienzo y terminación */
   int cont, n = 23;
   long int ciclo, maxciclo = 10000000;
   register int f, f1, f2;

   /* marcar el tiempo de comienzo */
   time(&prin);

   /* realizar múltiples bucles */
   for (ciclo = 1; ciclo <= maxciclo; ++ciclo) {
        f1 = 1;
        f2 = 1;

        /* generar los primeros n números de Fibonacci */
        for (cont = 1; cont <= n; ++cont) {
            f = (cont < 3) ? 1 : f1 + f2;
            f2 = f1;
            f1 = f;
        }
   }

   /* ajustar el contador y marcar el tiempo de terminación */
   --cont;
   time(&fin);

   /* mostrar la salida */
   printf("i = %d  F = %d\n", cont, f);
   printf("tiempo transcurrido: %.01f segundos", difftime(fin, prin));
}
```

Este programa incluye tres variables enteras que tienen el tipo de almacenamiento `register`. Sólo se calculan los 23 primeros números de Fibonacci, ya que éstos se representan como variables enteras ordinarias (números de Fibonacci más altos generarían un desbordamiento "overflow" de enteros).

El cálculo de los números de Fibonacci se repite 10 000 000 veces, para obtener una medida razonablemente acertada del tiempo de ejecución del programa. La única salida generada es el valor del número vigésimo tercero de Fibonacci, calculado al final del último ciclo. Por tanto, el programa es de cálculo intensivo (mínima entrada/salida) para enfatizar la ventaja usando el tipo de almacenamiento `register`.

Observe que el programa incluye su mecanismo de tiempo propio. En particular, el programa usa la función de biblioteca `time`, que asigna la hora actual (en segundos) a las variables `prin` y `fin`. Estas variables son del tipo `time_t`, definido en el archivo de cabecera `time.h`. El programa usa también la función de biblioteca `difftime`, que devuelve la diferencia de tiempo entre las dos variables `fin` y `prin`.

Ejecutando el programa en una computadora personal tipo pentium, se obtienen los siguientes resultados:

```
i = 23    F = 28657
tiempo transcurrido: 37 segundos
```

Si el programa se vuelve a ejecutar sin la declaración `register` (es decir, si las variables `f`, `f1` y `f2` son variables enteras ordinarias), la salida es esencialmente la misma. Por tanto, el uso del tipo `register` no proporciona una ventaja apreciable. Sin embargo, cuando se ejecuta en una computadora personal más antigua, el uso del tipo registro produce una reducción de un 36 por ciento en el tiempo empleado por la computadora. Los tamaños de los programas objeto compilados, con y sin variables `register`, no son significativamente diferentes con cada computadora.

Aunque el tipo de almacenamiento `register` se asocia normalmente con variables de tipo entero, algunos compiladores permiten que sea asociado con otros tipos de variables que tengan el mismo tamaño de palabra (por ejemplo, enteros `short` o `unsigned`). Además, se pueden permitir *punteros* a tales variables.

La especificación del tipo de almacenamiento `register` puede incluirse como una parte de la declaración formal de argumento dentro de una función, o como una parte de la especificación de tipo de argumento dentro de un prototipo de función. (*Observe que* `register` *es el único especificador de tipo de almacenamiento que puede usarse de esta manera*.)

EJEMPLO 13.3. El esquema de la estructura de un programa en C se muestra a continuación.

```
void func(register unsigned u, register int *pt);  /* prototipo de
                                                      función */

main()
{
   register unsigned u;      /* declaración de variable */
   register int *pt;         /* declaración de puntero */

   u = 5;                    /* asigna una cantidad entera */
   *pt = 12;                 /* asigna una cantidad entera */

   func(u, pt);
}

void func(register unsigned u, register int *pt) /* definición
                                                    de función */
{
   . . . . .
   return;
}
```

El prototipo de función indica que el primer argumento transferido a `func` es un entero sin signo que tiene el tipo de almacenamiento `register`, y el segundo argumento es un puntero a entero con el

mismo tipo de almacenamiento. Observe que las declaraciones formales de argumentos en `func` son consistentes con las especificaciones de argumentos que se muestran en la definición de la función.

Dentro de `main` vemos que `u` es un entero sin signo y `pt` un puntero a un entero. Ambas variables tienen asignado el especificador de tipo de almacenamiento `register`. Por tanto, `u` representará a un entero sin signo que se almacena en un registro de la unidad central de procesamiento de la computadora y `pt` apuntará al contenido de otro registro. En ambos casos, el uso de los registros de la computadora estará restringido a su disponibilidad.

A continuación de las declaraciones se asigna un valor de `5` a `u` y un valor de `12` a la posición a la que apunta `pt`. Estos valores se almacenarán en los registros de la computadora, siempre que éstos estén disponibles. Las variables `u` y `pt` se pasan a `func`, donde se procesan de forma no especificada.

13.2. OPERACIONES A NIVEL DE BITS

Algunas aplicaciones requieren la manipulación de los bits individuales en una palabra de memoria. Normalmente se requiere el lenguaje ensamblador o el lenguaje máquina para este tipo de operaciones. Sin embargo, C contiene varios operadores especiales que realizan fácil y eficientemente estas operaciones a nivel de bits. Estos operadores a nivel de bits se pueden dividir en tres categorías generales: el operador de complemento a uno, los operadores lógicos a nivel de bits y los operadores de desplazamiento. C también contiene algunos operadores que combinan las operaciones a nivel de bits con la asignación ordinaria. A continuación se discute cada categoría separadamente.

El operador de complemento a uno

El *operador de complemento a uno* (~) es un operador unario que invierte los bits de su operando, de modo que los unos se transforman en ceros y los ceros en unos. Este operador precede siempre a su operando. El operando tiene que ser un entero (incluyendo `integer`, `long`, `short`, `unsigned`, `char`, etc.). Generalmente el operando será un octal o una cantidad hexadecimal sin signo, pero no es obligatorio.

EJEMPLO 13.4. Consideremos la constante hexadecimal `0x7ff`. El patrón correspondiente de bits, expresado en términos de una palabra de 16 bits, es `0000 0111 1111 1111` (ver Apéndice A). El complemento a uno de este patrón de bits es `1111 1000 0000 0000`, que corresponde con el número hexadecimal `f800`. Por tanto, vemos que el valor de la expresión `~0x7ff` es `0xf800`. (Observe que los patrones de bits en este ejemplo se han ordenado en grupos de cuatro únicamente por conveniencia.)

A continuación se muestran algunas expresiones que usan el operador de complemento a uno y sus correspondientes valores. Todos los resultados se expresan en términos de una palabra de 16 bits.

Expresión	*Valor*	
`~0xC5`	`0xff3a`	(constantes hexadecimales)
`~0x1111`	`0xeeee`	(constantes hexadecimales)
`~0xffff`	`0`	(constantes hexadecimales)
`~052`	`0177725`	(constantes octales)
`~0177777`	`0`	(constantes octales)

En las dos últimas expresiones, el dígito octal del extremo izquierdo equivale a un solo bit (de otra forma el patrón excedería de 16 bits).

Se anima al lector a demostrar la validez de estas expresiones escribiendo los patrones correspondientes de bits como se ha mostrado con anterioridad.

El operador de complemento a uno se refiere a veces como *operador de complementación*. Es un miembro del mismo grupo de precedencia que los otros operadores unarios. Por tanto, su asociatividad es de derecha a izquierda.

EJEMPLO 13.5. Consideremos el programa en C que se muestra a continuación.

```
#include <stdio.h>

main()
{
  unsigned i = 0x5b3c;

  printf("valores hexadecimales:  i = %x   ~i = %x\n", i, ~i);
  printf("decimales equivalentes: i = %u   ~i = %u", i, ~i);
}
```

La ejecución de este programa en una computadora con tamaño de palabra de 16 bits produce la siguiente salida:

```
valores hexadecimales:  i = 5b3c   ~i = a4c3
decimales equivalentes: i = 23356   ~i = 42179
```

Para entender estos resultados, consideremos primero los patrones de bits correspondientes a los valores de `i` y `~i`.

```
 i = 0101 1011 0011 1100
~i = 1010 0100 1100 0011
```

El decimal equivalente de este primer patrón de bits se puede determinar escribiendo

$$i = 0\text{x}2^{15} + 1\text{x}2^{14} + 0\text{x}2^{13} + 1\text{x}2^{12} + 1\text{x}2^{11} + 0\text{x}2^{10} + 1\text{x}2^{9} + 1\text{x}2^{8} + 0\text{x}2^{7} + 0\text{x}2^{6} + 1\text{x}2^{5} + 1\text{x}2^{4} + 1\text{x}2^{3} + 1\text{x}2^{2} + 0\text{x}2^{1} + 0\text{x}2^{0} =$$

$$16384 + 4096 + 2048 + 512 + 256 + 32 + 16 + 8 + 4 = 23356$$

Por tanto, el decimal equivalente de `0x5b3c` es `23356`.

Análogamente, el equivalente decimal del segundo patrón se puede determinar escribiendo

$$\sim i = 1\text{x}2^{15} + 0\text{x}2^{14} + 1\text{x}2^{13} + 0\text{x}2^{12} + 0\text{x}2^{11} + 1\text{x}2^{10} + 0\text{x}2^{9} + 0\text{x}2^{8} + 1\text{x}2^{7} + 1\text{x}2^{6} + 0\text{x}2^{5} + 0\text{x}2^{4} + 0\text{x}2^{3} + 0\text{x}2^{2} + 1\text{x}2^{1} + 1\text{x}2^{0} =$$

$$32768 + 8192 + 1024 + 128 + 64 + 2 + 1 = 42179$$

Por tanto, vemos que el equivalente decimal de `0xa4c3` es `42179`.

Operadores lógicos a nivel de bits

Hay tres operadores lógicos a nivel de bits: *y a nivel de bits* (&), *o exclusiva a nivel de bits* (^), y *o a nivel de bits* (|). Cada uno de estos operadores requiere dos operandos enteros. Las operaciones se realizan de forma independiente en cada par de bits que corresponden a cada operando. Así, se compararán los bits menos significativos (los bits del extremo derecho) de las dos palabras, después los siguientes bits menos significativos, y así sucesivamente, hasta que se comparen todos los bits. Los resultados de estas comparaciones son:

- Una expresión *y a nivel de bits* devolverá un 1 si ambos bits tienen el valor 1 (si ambos bits son verdaderos). En otro caso devolverá un valor de 0.
- Una expresión *o exclusiva a nivel de bits* devolverá un 1 si uno de los dos bits tiene un valor de 1 y el otro tiene un valor de 0 (un bit es verdadero y el otro es falso). En otro caso devolverá un valor de 0.
- Una expresión *o a nivel de bits* devolverá un 1 si uno o más de los bits tiene el valor de 1 (uno o ambos son verdaderos). En otro caso devolverá un valor de 0.

Estos resultados están resumidos en la Tabla 13.1. En esta tabla, *b1* y *b2* representan los bits correspondientes dentro del primer y segundo operando, respectivamente.

Tabla 13.1. Operaciones lógicas a nivel de bits

b1	*b2*	*b1 & b2*	*b1 ^ b2*	*b1 | b2*
1	1	1	0	1
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0

EJEMPLO 13.6. Supongamos que a y b son variables enteras sin signo cuyos valores son 0x6db7 y 0xa726, respectivamente. A continuación se muestran los resultados de varias operaciones a nivel de bits con estas variables.

```
~a = 0x9248

~b = 0x58d9

a & b = 0x2526

a ^ b = 0xca91

a | b = 0xefb7
```

La validez de estas expresiones puede verificarse expandiendo cada uno de los patrones de bits. De esta manera,

```
      a = 0110 1101 1011 0111
          ___________________
     ~a = 1001 0010 0100 1000
        = 0x9248


      b = 1010 0111 0010 0110
          ___________________
     ~b = 0101 1000 1101 1001
        = 0x58d9


      a = 0110 1101 1011 0111
      b = 1010 0111 0010 0110
          ___________________
  a & b = 0010 0101 0010 0110
        = 0x2526


      a = 0110 1101 1011 0111
      b = 1010 0111 0010 0110
          ___________________
  a ^ b = 1100 1010 1001 0001
        = 0xca91


      a = 0110 1101 1011 0111
      b = 1010 0111 0010 0110
          ___________________
  a | b = 1110 1111 1011 0111
        = 0xefb7
```

Cada uno de los operadores lógicos a nivel de bits tiene su propia precedencia. El operador *y a nivel de bits* (`&`) tiene la mayor precedencia, seguido por la *o exclusiva a nivel de bits* (`^`), después la *o a nivel de bits* (`|`). La *y a nivel de bits* sigue a los operadores de igualdad (`==` y `!=`). La *o a nivel de bits* es seguida por la *y lógica* (`&&`). La asociatividad para cada operador a nivel de bits es de izquierda a derecha. (Ver Apéndice C para un resumen de todos los operadores de C, con sus precedencias y sus asociatividades.)

Enmascaramiento

El *enmascaramiento* es un proceso en el que un patrón dado de bits se convierte en otro patrón por medio de una operación lógica a nivel de bits. El patrón original de bits es uno de los operandos en la operación a nivel de bits. El segundo operando, llamado *máscara*, es un patrón especialmente escogido que realiza la transformación deseada.

Hay varios tipos distintos de operaciones de enmascaramiento. Por ejemplo, una parte del patrón dado puede copiarse a una nueva palabra rellenando el resto de la nueva palabra con ceros. Así, una parte del patrón original será «enmascarada» del resultado final. El operador *y a*

nivel de bits (&) se usa para este tipo de operaciones de enmascaramiento, como se ilustra a continuación.

EJEMPLO 13.7. Supongamos que a es una variable entera sin signo cuyo valor es 0x6db7. Extraeremos los 6 bits de la derecha de este valor y los asignaremos a la variable entera sin signo b. Asignaremos ceros a los 10 bits de b de la izquierda.

Para efectuar esta operación, escribiremos la expresión a nivel de bits

```
b = a & 0x3f;
```

El segundo operando, la constante hexadecimal 0x3f, servirá de máscara. Así, el valor resultante de b será 0x37.

La validez de este resultado puede establecerse examinando el patrón de bits correspondiente.

```
       a = 0110  1101  1011  0111
máscara = 0000  0000  0011  1111
          ______________________
       b = 0000  0000  0011  0111
         = 0x37
```

Observe que la máscara impide que los 10 bits de la izquierda se copien de a a b.

La máscara en este último ejemplo contiene unos en las posiciones de la derecha (las posiciones de los bits menos significativos) y ceros en las posiciones de la izquierda (las posiciones de los bits más significativos). Tales máscaras son independientes de la longitud de la palabra, ya que se usan ceros para rellenar las posiciones restantes una vez que los unos requeridos han sido colocados en las posiciones de menor orden. Si se requieren los unos en las posiciones de la derecha, la máscara tendrá que estar relacionada con la longitud de la palabra. (Recordar que la posición del bit del extremo derecho siempre representa 2^0, mientras que el bit de la posición del extremo izquierdo representa 2^{n-1}, donde n es el número de bits de la palabra.) Sin embargo, tal dependencia puede evitarse a menudo escribiendo la máscara en términos de su complemento a uno.

EJEMPLO 13.8. Supongamos de nuevo que a es una variable entera sin signo cuyo valor es 0x6db7. Extraeremos ahora los 6 bits *de la izquierda* de este valor y los asignaremos a la variable entera sin signo b. Asignaremos ceros a los 10 bits de la derecha de b.

Para efectuar esta operación podemos escribir la expresión a nivel de bits

```
b = a & 0xfc00;
```

Así, la constante hexadecimal 0xfc00 servirá de máscara. El valor resultante de b será 0x6c00.

La validez de este resultado puede establecerse de nuevo examinando los patrones de bits correspondientes.

```
       a = 0110  1101  1011  0111
máscara = 1111  1100  0000  0000
          ______________________
       b = 0110  1100  0000  0000
         = 0x6c00
```

La máscara bloquea ahora los 10 bits de la derecha en a.

La máscara, en esta situación, depende del tamaño de palabra de 16 bits, ya que los unos aparecen en las posiciones de la izquierda. Sin embargo, si la máscara se escribe en términos de su complemento a uno, los unos aparecen en las posiciones de los bits de la derecha y el resto de las posiciones están llenas de ceros. Por tanto, la máscara se vuelve independiente del tamaño de la palabra.

El complemento a uno de la máscara original es la constante hexadecimal `0x3ff`. Por tanto, podemos expresar este enmascaramiento como

```
b = a & ~0x3ff;
```

El valor resultante de `b` será como antes `0x6c00`.

La validez de este resultado puede verse examinando los correspondientes patrones de bits mostrados a continuación.

```
 0x3ff = 0000 0011 1111 1111
         ___________________
~0x3ff = 1111 1100 0000 0000 = 0xfc00 (la máscara original)

     a = 0110 1101 1011 0111
~0x3ff = 1111 1100 0000 0000
         ___________________
     b = 0110 1100 0000 0000
       = 0x6c00
```

Otro tipo de operación de enmascaramiento permite copiar una parte de un patrón de bits dado a una nueva palabra, rellenando el resto de la palabra con unos. Para hacerlo se utiliza el operador *o a nivel de bits*. (Observe la distinción entre esta operación de enmascaramiento y la anterior, que permitía copiar una parte de un patrón de bits a una nueva palabra, mientras el resto de la nueva palabra se rellena con ceros.)

EJEMPLO 13.9. Supongamos como antes que `a` es una variable entera sin signo cuyo valor es `0x6db7`, como antes. Transformaremos el patrón correspondiente de bits en otro en el cual los 8 bits de la derecha sean unos y los 8 bits de la izquierda conserven su valor original. Asignaremos este nuevo patrón a la variable entera sin signo `b`.

Esta operación se realiza mediante la expresión a nivel de bits

```
b = a | 0xff;
```

La constante hexadecimal `0xff` es la máscara. El valor resultante de `b` será `0x6dff`.

Examinaremos los correspondientes patrones de bits, para verificar la corrección del resultado.

```
      a = 0110 1101 1011 0111
máscara = 0000 0000 1111 1111
          ___________________
      b = 0110 1101 1111 1111
        = 0x6dff
```

Recordar que la operación a nivel de bits es ahora la *o a nivel de bits* y no la *y a nivel de bits*, como en los ejemplos anteriores. Así, cuando cada uno de los 8 bits de la derecha en `a` se compara con el correspondiente

1 en la máscara, el resultado es siempre 1. Sin embargo, cuando cada uno de los 8 bits de la izquierda en a se compara con el correspondiente 0 en la máscara, el resultado será el mismo que el bit original en a.

Supongamos ahora que queremos transformar el patrón de bits de a en otro en el cual los 8 bits de *la izquierda* sean todos unos y los 8 bits de *la derecha* conserven su valor original. Esto se puede realizar mediante cualquiera de las siguientes dos expresiones:

```
b = a | 0xff00;
```

o

```
b = a | ~0xff;
```

En cualquier caso, el valor resultante de b será 0xffb7. Resulta preferible la segunda expresión, ya que es independiente del tamaño de palabra.

El lector debería verificar la corrección de estos resultados expandiendo los correspondientes patrones de bits y realizando las operaciones indicadas.

Se puede copiar una parte de un patrón dado de bits a una nueva palabra, mientras que el resto del patrón de bits original se invierten dentro de la nueva palabra. Este tipo de enmascaramiento utiliza la *o exclusiva a nivel de bits*. Los detalles se ilustran en el siguiente ejemplo.

EJEMPLO 13.10. Supongamos que a es una variable entera sin signo cuyo valor es 0x6db7, como en los últimos ejemplos. Invirtamos los 8 bits de la derecha y preservemos los 8 bits de la izquierda. Este nuevo patrón de bits se asignará a la variable entera sin signo b.

Para ejecutarlo haremos uso de la operación *o exclusiva a nivel de bits*.

```
b = a ^ 0xff;
```

La constante hexadecimal 0xff es la máscara. Esta expresión asignará el valor 0x6d48 a b.

Aquí tenemos los correspondientes patrones de bits.

```
        a = 0110 1101 1011 0111
  máscara = 0000 0000 1111 1111
            -------------------
        b = 0110 1101 0100 1000
          = 0x6d48
```

Recordar que ahora la operación a nivel de bits es la *o exclusiva a nivel de bits* en vez de la *y* o la *o a nivel de bits*. Por tanto, cuando cada uno de los 8 bits de la derecha en a se compara con el correspondiente 1 de la máscara, el bit resultante será el opuesto al bit original de a. Por otra parte, cuando cada uno de los 8 bits de la izquierda se compara con el correspondiente 0 de la máscara, el bit resultante será el mismo que el bit original en a.

Si se quieren invertir los 8 bits de la izquierda en a mientras se preservan los 8 bits originales de la derecha, podemos escribir

```
b = a ^ 0xff00;
```

o la expresión más deseable (ya que es independiente del tamaño de palabra)

```
b = a ^ ~0xff;
```

El valor resultante de cada expresión es 0x92b7.

La operación *o exclusiva* puede usarse repetidamente como un *conmutador* para cambiar el valor de un bit particular dentro de una palabra. En otras palabras, si un bit tiene un valor de `1`, la operación *o exclusiva* cambiará su valor a `0`, y viceversa. Tales operaciones son particularmente comunes en programas que interactúan estrechamente con el hardware de la computadora.

EJEMPLO 13.11. Supongamos que `a` es una variable entera sin signo cuyo valor es `0x6db7`, como en los últimos ejemplos. La expresión

```
a ^ 0x4
```

invertirá el valor del bit número `2` (el tercer bit por la derecha) dentro de `a`. Si esta operación se realiza repetidamente, el valor de `a` alternará entre `0x6db7` y `0x6db3`. Así, usando esta operación repetidamente conmutará el tercer bit por la derecha entre uno y cero.

Los patrones de bits correspondientes se muestran a continuación.

```
 0x6db7 = 0110 1101 1011 0111
máscara = 0000 0000 0000 0100
          -------------------
 0x6db3 = 0110 1101 1011 0011
máscara = 0000 0000 0000 0100
          -------------------
 0x6db7 = 0110 1101 1011 0111
```

Los operadores de desplazamiento

Los dos operadores de desplazamiento a nivel de bits son *desplazamiento a la izquierda* (<<) y *desplazamiento a la derecha* (>>). Cada operador requiere dos operandos. El primero es un operando de tipo entero que representa el patrón de bits a desplazar. El segundo es un entero sin signo que indica el número de desplazamientos (si los bits en el primer operando son desplazados en 1 posición de bit, 2 posiciones de bit, 3 posiciones de bit, etc.). Este valor no puede exceder del número de bits asociado con el tamaño de palabra del primer operando.

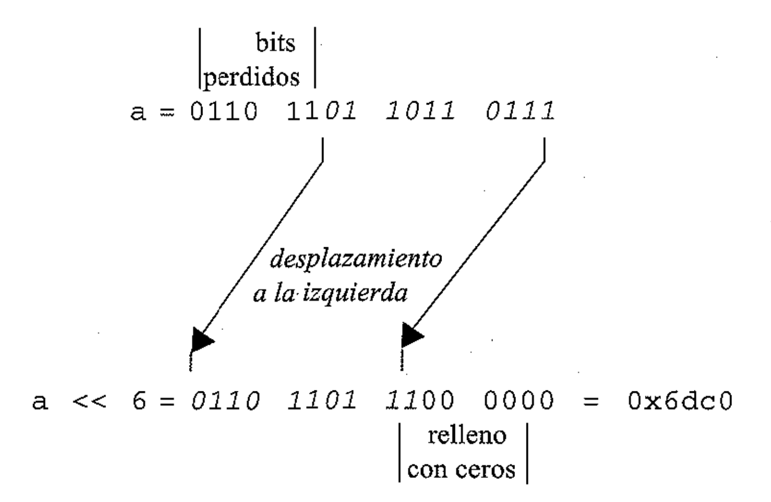
El operador de desplazamiento a la izquierda hace que los bits en el primer operando sean desplazados a la izquierda el número de posiciones indicado por el segundo operando. Se perderán los bits de la izquierda (los bits de desbordamiento) en el patrón original de bits. Las posiciones de la derecha que quedan vacantes se rellenan con ceros.

EJEMPLO 13.12. Supongamos que `a` es una variable entera sin signo cuyo valor es `0x6db7`. La expresión

```
b = a << 6;
```

desplazará todos los bits seis posiciones a la izquierda y asignará el valor resultante a la variable entera sin signo `b`. El valor resultante de `b` será `0x6dc0`.

Escribamos los correspondientes patrones de bits para ver cómo se obtiene el resultado final.



Todos los bits asignados inicialmente a `a` se desplazan seis posiciones a la izquierda como indican las flechas. Los 6 bits de la izquierda (originalmente `0110 11`) se pierden. Las seis posiciones de la derecha se llenan con `00 0000`.

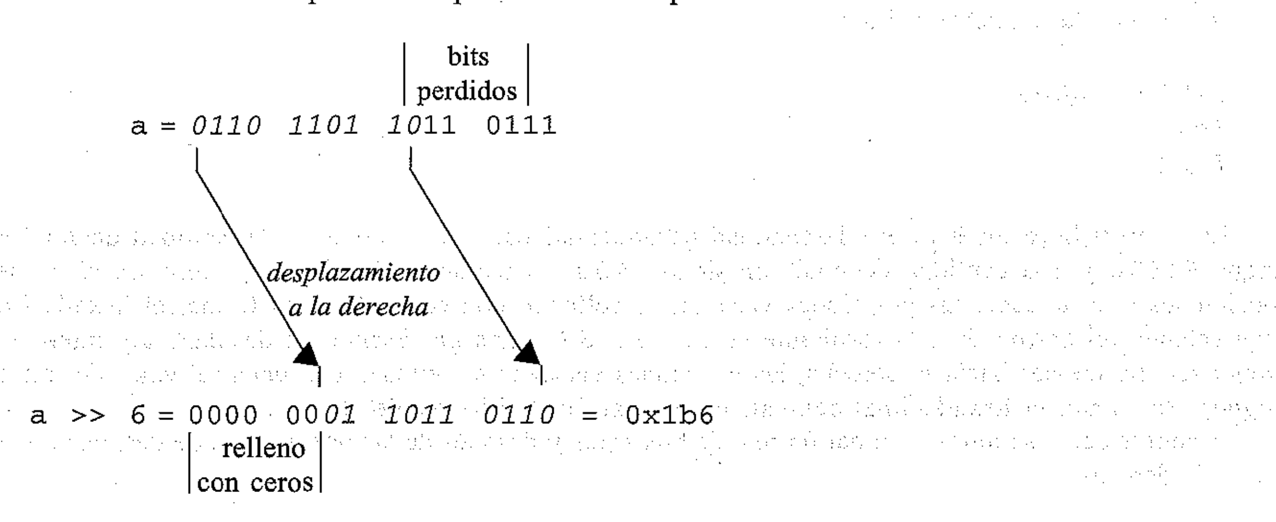
El operador de desplazamiento a la derecha hace que los bits en el primer operando sean desplazados a la derecha el número de posiciones indicadas por el segundo operando. Se perderán los bits de la derecha (los bits de «underflow») en el patrón original de bits. Si el patrón de bits que se desplaza representa un entero *sin signo*, las posiciones de la izquierda que quedan vacantes se rellenan con ceros. Por tanto, el comportamiento del operador desplazamiento a la derecha es similar al de desplazamiento a la izquierda cuando el primer operando es un entero sin signo.

EJEMPLO 13.13. Supongamos que `a` es una variable entera sin signo cuyo valor es `0x6db7`. La expresión

```
b = a >> 6;
```

desplazará todos los bits seis posiciones a la derecha y asignará el valor resultante a la variable entera sin signo `b`. El valor resultante de `b` será `0x1b6`.

Escribamos los correspondientes patrones de bits para ver cómo se obtiene el resultado final.



Vemos que todos los bits asignados inicialmente a a se desplazan seis posiciones a la derecha como indican las flechas. Los 6 bits de la derecha (originalmente 11 0111) se pierden. Las seis posiciones de la izquierda se llenan con 00 0000.

Si se desplaza a la derecha el patrón de bits correspondiente a un entero *con signo*, el resultado puede depender del valor del bit más a la izquierda (el bit del signo). La mayoría de los compiladores llenan las posiciones vacantes con el contenido de este bit. (Los valores negativos tendrán este bit a 1, mientras que los positivos lo tendrán a 0.) Sin embargo, otros compiladores rellenarán las posiciones vacantes con ceros independientemente del bit de signo del entero original. Debería determinar cómo maneja esta situación su compilador particular.

EJEMPLO 13.14. A continuación se muestra un programa sencillo en C que ilustra el uso del operador desplazamiento a la derecha.

```
#include <stdio.h>

main()
{
   unsigned a = 0xf05a;
   int b = a;

   printf("%u %d\n", a, b);
   printf("%x\n", a >> 6);
   printf("%x\n", b >> 6);
}
```

Observe que a representa una cantidad entera sin signo, mientras que b representa una cantidad entera ordinaria (con signo). Ambas variables tienen asignado originalmente el valor (hexadecimal) 0xf05a. Como la posición del extremo izquierdo contiene un 1, el entero con signo (b) interpretará este valor como un número negativo.

El programa muestra los valores decimales representados por los patrones de bits asignados a a y b. Después se ve el resultado de desplazar seis posiciones hacia la derecha cada cantidad. Así, si el programa se ejecuta en un compilador que copia el contenido del bit de signo en las posiciones vacantes, se obtiene la siguiente salida:

```
61530 -4006
3c1
ffc1
```

La primera línea muestra que la cantidad hexadecimal 0xf05a equivale a la cantidad decimal sin signo 61530 y a la cantidad decimal con signo -4006. Cuando se desplaza el entero *sin signo* seis posiciones a la derecha, las posiciones vacantes se rellenan con ceros. De esta forma, el hexadecimal equivalente del patrón de bits resultante es 0x3c1. Sin embargo, cuando se desplaza un entero *con signo* seis posiciones hacia la derecha, las posiciones vacantes se rellenan con unos (el valor del bit del signo). Por tanto, el hexadecimal equivalente del patrón de bits resultante es ffc1.

A continuación se muestra el patrón real de bits antes y después de las operaciones de desplazamiento hacia la derecha.

```
        a = 1111 0000 0101 1010

   a >> 6 = 0000 0011 1100 0001 = 0x3c1

        b = 1111 0000 0101 1010

   b >> 6 = 1111 1111 1100 0001 = 0xffc1
```

Los operadores de asignación a nivel de bits

C también contiene los siguientes operadores de *asignación a nivel de bits*:

```
&=      ^=      |=      <<=     >>=
```

Estos operadores combinan las operaciones precedentes a nivel de bits con la asignación. El operando de la izquierda debe ser un identificador de variable de tipo entero (por ejemplo, una variable entera), y el operando de la derecha debe ser una expresión a nivel de bits. El operando de la izquierda se interpreta como el primer operando en la expresión a nivel de bits. El valor de la expresión a nivel de bits se asigna al operando de la izquierda. Por ejemplo, la expresión `a &= 0x7f` es equivalente a `a = a & 0x7f`.

Los operadores de asignación a nivel de bits son miembros del mismo grupo de precedencia que el resto de los operadores de asignación. Su asociatividad es de derecha a izquierda (ver Apéndice C).

EJEMPLO 13.15. A continuación se muestran varias expresiones de asignación a nivel de bits. En cada expresión se supone que `a` es una variable entera sin signo cuyo valor inicial es `0x6db7`.

Expresión	*Expresión equivalente*	*Valor final*
`a &= 0x7f`	`a = a & 0x7f`	`0x37`
`a ^= 0x7f`	`a = a ^ 0x7f`	`0x6dc8`
`a |= 0x7f`	`a = a | 0x7f`	`0x6dff`
`a <<= 5`	`a = a << 5`	`0xb6e0`
`a >>= 5`	`a = a >> 5`	`0x36d`

Muchas aplicaciones involucran el uso de múltiples operaciones a nivel de bits. De hecho aparecen a menudo dos o más operaciones a nivel de bits en la misma expresión.

EJEMPLO 13.16. Presentación de patrones de bits. La mayoría de las versiones de C no incluyen una función de biblioteca para convertir un entero decimal en su patrón de bits correspondiente. A continuación se muestra un programa completo en C para realizar esta conversión. El programa mostrará el patrón correspondiente de bits para una cantidad entera positiva o negativa.

```
/* mostrar el patrón de bits correspondiente a un entero decimal con
   signo */

#include <stdio.h>
```

```
main()
{
   int a, b, m, cont, nbits;
   unsigned mascara;

   /* determinar el tamaño de palabra en bits y poner la máscara
      inicial */
   nbits = 8 * sizeof(int);
   m = 0x1 << (nbits - 1);    /* colocar un 1 en la posición de la
                                 izquierda */

   /* bucle principal */
   do {
        /* leer un entero con signo */
        printf("\n\nIntroducir un valor entero (0 para parar): ");
        scanf("%d", &a);

        /* salida del patrón de bits */
        mascara = m;
        for (cont = 1; cont <= nbits; cont++) {
             b = (a & mascara) ? 1 : 0; /* poner el bit a mostrar
                                           a 1 o 0 */
             printf("%x", b);           /* escribir el bit */
             if (cont % 4 == 0)
                 printf(" ");           /* espacio en blanco cada
                                           cuatro dígitos */
             mascara >>= 1;             /* desplazar la máscara
                                           una posición a la de-
                                           recha */
        }
   } while (a != 0);
}
```

El programa se ha escrito de modo que es independiente del tamaño de palabra para un entero. Por tanto se puede utilizar en cualquier computadora. Empieza determinando el tamaño de palabra en bits. Después asigna un valor inicial apropiado a la variable entera `m`. Este valor se usará como máscara en una operación *y a nivel de bits*. Observe que `m` contiene un `1` en la posición del extremo izquierdo y ceros en el resto de las posiciones.

La parte principal del programa es un bucle `do-while` que permite convertir múltiples cantidades enteras en sus patrones de bits equivalentes. Cada paso a través del bucle introduce un entero en la computadora, lo convierte en su patrón equivalente de bits y lo muestra. La ejecución continúa hasta que se introduce el valor `0` y se convierte en una sucesión de ceros.

Cuando se ha introducido el entero, se le asigna a la máscara el valor inicial definido al principio del programa. Se usa un bucle `for` para examinar el entero bit a bit, empezando por el más significativo (el bit del extremo izquierdo). Para examinar cada posición se usa una operación de enmascaramiento basada en el uso de la *y a nivel de bits*. Después se muestra el contenido de cada posición. Finalmente, el `1` de la máscara se desplaza una posición a la derecha, como anticipación para examinar el siguiente bit.

Observe que todos los bits se muestran en la misma línea. Para aumentar la legibilidad de la salida se introduce un espacio en blanco después de cada grupo de cuatro bits.

A continuación se muestra el típico diálogo interactivo resultante de la ejecución del programa. Las respuestas del usuario están subrayadas.

```
Introducir un valor entero (0 para parar): 1
0000 0000 0000 0001

Introducir un valor entero (0 para parar): -1
1111 1111 1111 1111

Introducir un valor entero (0 para parar): 129
0000 0000 1000 0001

Introducir un valor entero (0 para parar): -129
1111 1111 0111 1111

Introducir un valor entero (0 para parar): 1024
0000 0100 0000 0000

Introducir un valor entero (0 para parar): -1024
1111 1100 0000 0000

Introducir un valor entero (0 para parar): 7033
0001 1011 0111 1001

Introducir un valor entero (0 para parar): -7033
1110 0100 1000 0111

Introducir un valor entero (0 para parar): 32767
0111 1111 1111 1111

Introducir un valor entero (0 para parar): -32768
1000 0000 0000 0000

Introducir un valor entero (0 para parar): 0
0000 0000 0000 0000
```

Observe que cada número positivo tiene un 0 en la posición del extremo izquierdo, y cada número negativo tiene un 1 en esa posición. (En realidad, el patrón de bits de un número negativo es el *complemento a dos* del patrón de bits para un número positivo. Para obtener el complemento a dos, se hace el complemento a uno y se le suma 1 al bit del extremo derecho.)

13.3. CAMPOS DE BITS

En algunas aplicaciones se requiere trabajar con elementos que consten de unos pocos bits (por ejemplo un indicador de un bit que indique una condición verdadera/falsa, un entero de 3 bits cuyo rango vaya de 0 a 7, o un carácter ASCII de 7 bits). Varios datos de este tipo se pueden empaquetar en una sola palabra de memoria. Para hacer esto, la palabra se subdivide en *campos de bits* individuales. Estos campos de bits se definen como miembros de una estructura. Cada campo de bits puede ser accedido individualmente, como cualquier otro miembro de una estructura.

En términos generales, la descomposición de una palabra en distintos campos de bits puede escribirse como

```
struct  marca   {
                miembro 1;
                miembro 2;
                . . . . .
                miembro m;
};
```

donde los elementos individuales tienen el mismo significado que en una declaración de estructura. Cada declaración de miembro debe incluir ahora una especificación que indique el tamaño del campo de bits correspondiente. Para hacerlo, el nombre del miembro debe ir seguido por dos puntos y un entero sin signo que indique el tamaño del campo.

La interpretación de estos campos de bits puede variar de un compilador a otro. Por ejemplo, algunos compiladores pueden ordenar los campos de bits de derecha a izquierda, mientras que otros los ordenan de izquierda a derecha. Supondremos ordenación de derecha a izquierda en los ejemplos mostrados a continuación.

EJEMPLO 13.17. Un programa en C contiene las siguientes declaraciones:

```
struct   muestra   {
                   unsigned a  :  1;
                   unsigned b  :  3;
                   unsigned c  :  2;
                   unsigned d  :  1;
};

struct   muestra   v;
```

La primera declaración define una estructura que se subdivide en cuatro campos de bits, llamados `a`, `b`, `c` y `d`. Estos campos de bits tienen tamaños de `1` bit, `3` bits, `2` bits y `1` bit, respectivamente. Así, los campos de bits ocupan un total de `7` bits dentro de una palabra de memoria. El resto de los bits dentro de la palabra quedarán sin usar.

La Figura 13.1 ilustra el esquema de los campos de bits dentro de la palabra, suponiendo una palabra de `16` bits y ordenación de derecha a izquierda.

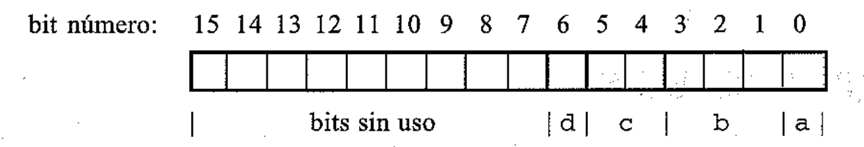


Figura 13.1. Campos de bits dentro de una palabra de 16 bits.

La segunda declaración establece que `v` es una variable de estructura del tipo `muestra`. Así, `v.a` es un campo dentro de `v` cuyo tamaño es `1` bit. Análogamente, `v.b` es un campo cuyo tamaño es `3` bits; y así sucesivamente.

Un campo de bits sólo puede definirse como una parte de una palabra `integer` o `unsigned`. (Algunos compiladores también permiten que un campo de bits sea una parte de una palabra `char` o `long`.) Sin embargo, en todos los demás aspectos, las reglas para definir campos de bits son las mismas que las que gobiernan otros tipos de estructuras.

EJEMPLO 13.18. Las declaraciones del Ejemplo 13.17 pueden agruparse

```
struct  muestra  {
                  unsigned a : 1;
                  unsigned b : 3;
                  unsigned c : 2;
                  unsigned d : 1;
}  v;
```

La interpretación de la variable v es la misma que la dada en el Ejemplo 13.17. Además, la marca puede omitirse de modo que la declaración anterior puede reducirse aún más a

```
struct  {
          unsigned a : 1;
          unsigned b : 3;
          unsigned c : 2;
          unsigned d : 1;
}  v;
```

Un campo dentro de una estructura no puede sobrepasar una palabra dentro de la memoria de la computadora. Este problema no surge si la suma de los tamaños de los campos no excede del tamaño de una cantidad entera sin signo. Sin embargo, si la suma de los tamaños de los campos excede este tamaño de palabra, cualquiera que lo sobrepase será forzado automáticamente al principio de la siguiente palabra.

EJEMPLO 13.19. Consideremos el programa sencillo en C mostrado a continuación.

```
#include <stdio.h>

main()
{
  static struct {
                unsigned a : 5;  /* principio primera palabra */
                unsigned b : 5;
                unsigned c : 5;
                unsigned d : 5;  /* forzado a la segunda palabra */
  } v = {1, 2, 3, 4};

  printf("v.a = %d   v.b = %d   v.c = %d   v.d = %d\n", v.a, v.b, v.c,
          v.d);
  printf("v requiere %d bytes\n", sizeof(v));
}
```

Los cuatro campos de v requieren un total de 20 bits. Si la computadora sólo permite 16 bits para una cantidad entera sin signo, esta declaración de estructura requerirá dos palabras de memoria. Los tres primeros campos se almacenarán en la primera palabra. Como el último campo de bits sobrepasa el límite de la palabra, se fuerza automáticamente al principio de la segunda palabra.

La Figura 13.2 muestra el esquema de los campos de bits dentro de las dos palabras de `16` bits.

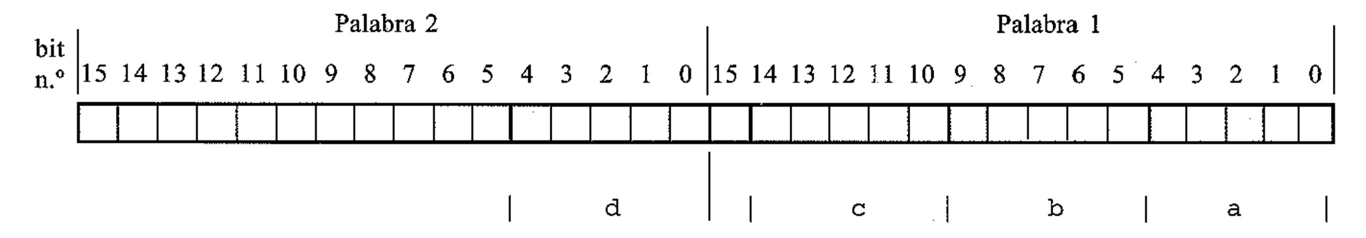


Figura 13.2. Cuatro campos de bits dentro de dos palabras de 16 bits.

La ejecución de este programa producirá la siguiente salida:

```
v.a = 1   v.b = 2   v.c = 3   v.d = 4
v requiere 4 bytes
```

La segunda línea verifica la necesidad de dos palabras, ya que cada palabra es equivalente a dos bytes. (Con algunos compiladores, `v` sólo necesitará `3` bytes, es decir, `24` bits.)

Los campos sin nombre se pueden usar para controlar la colocación de los campos de bits dentro de una palabra de memoria. Tales campos sirven de relleno dentro de la palabra. El tamaño de los campos sin nombre determina la extensión del relleno.

EJEMPLO 13.20. Consideremos el sencillo programa en C mostrado a continuación.

```
#include <stdio.h>

main()
{
   static struct {
                       unsigned a : 5;
                       unsigned b : 5;
                       unsigned c : 5;
   } v = {1, 2, 3};

   printf("v.a = %d   v.b = %d   v.c = %d\n", v.a, v.b, v.c);
   printf("v requiere %d bytes\n", sizeof(v));
}
```

Este programa es similar al mostrado en el ejemplo anterior. Sin embargo, ahora sólo se definen tres campos (`15` bits) dentro de `v`. Por tanto, sólo se requiere una palabra de memoria para almacenar esta estructura.

Ejecutando el programa se produce la siguiente salida:

```
v.a = 1   v.b = 2   v.c = 3
v requiere 2 bytes
```

La segunda línea de salida verifica que los tres campos pueden almacenarse en una palabra simple sin signo (2 bytes).

Modifiquemos el programa añadiendo un campo sin nombre cuyo tamaño es de 6 bits.

```
#include <stdio.h>

main()
{
  static struct {
                unsigned a : 5;   /* principio primera palabra */
                unsigned b : 5;
                unsigned   : 6;   /* rellenar primera palabra */
                unsigned c : 5;   /* principio segunda palabra */
  } v = {1, 2, 3};

  printf("v.a = %d   v.b = %d   v.c = %d\n", v.a, v.b, v.c);
  printf("v requiere %d bytes\n", sizeof(v));
}
```

Ahora se requerirán dos palabras de memoria. Los dos primeros campos se almacenarán dentro de la primera palabra, seguidos por 6 bits libres (para un total de 16 bits, rellenando así la primera palabra). Por tanto, el último campo se alineará con el principio de la segunda palabra, como se ilustra en la Figura 13.3.

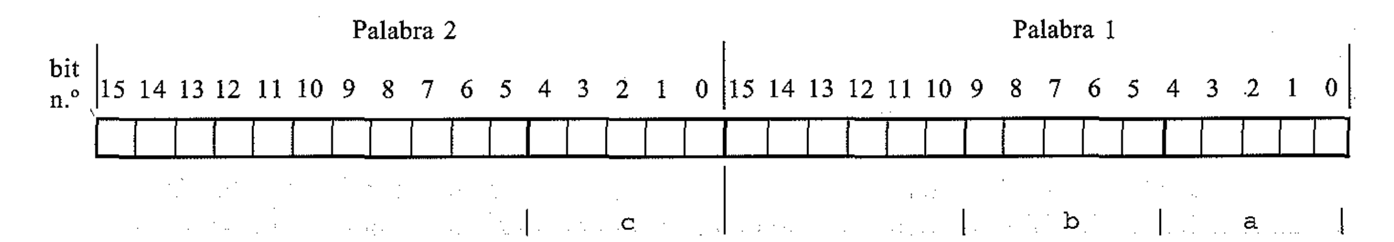


Figura 13.3. Tres campos de bits dentro de dos palabras de 16 bits.

Cuando se ejecuta este programa, se produce la siguiente salida:

```
v.a = 1   v.b = 2   v.c = 3
v requiere 4 bytes
```

De la segunda línea de la salida vemos que se requieren dos palabras (4 bytes) para almacenar los tres campos debido al relleno adicional.

Otra forma de controlar la alineación de los campos de bits es incluir un campo de bits sin nombre cuyo tamaño sea cero. Esto forzará automáticamente que el siguiente campo se alinee con el principio de una nueva palabra.

EJEMPLO 13.21. Consideremos el sencillo programa en C mostrado a continuación.

```
#include <stdio.h>

main()
{
  static struct {
                    unsigned a  :  5;/* principio primera palabra */
                    unsigned b  :  5;
                    unsigned    :  0;/* forzar alineamiento con la
                                        segunda palabra */
                    unsigned c  :  5;/* principio segunda palabra */
  } v = {1, 2, 3};

  printf("v.a = %d   v.b = %d   v.c = %d\n", v.a, v.b, v.c);
  printf("v requiere %d bytes\n", sizeof(v));
}
```

Este programa es similar al segundo mostrado en el último ejemplo. Sin embargo, ahora la declaración de estructura incluye un campo de bits sin nombre cuyo tamaño es cero. Esto forzará automáticamente que el último campo se coloque al principio de la nueva palabra, como se ilustró en la Figura 13.3.

Cuando se ejecuta el programa, se genera la siguiente salida:

```
v.a = 1   v.b = 2   v.c = 3
v requiere 4 bytes
```

La última línea verifica que para almacenar los tres campos se requieren dos palabras (4 bytes), como se definió anteriormente. (Con algunos compiladores, `v` requerirá sólo 3 bytes, esto es, 24 bits.)

Se recuerda de nuevo al lector que algunos compiladores ordenan los campos de bits de derecha a izquierda (bits de menor a mayor orden) dentro de una palabra, mientras que otros ordenan los campos de izquierda a derecha (bits de mayor a menor orden). Comprobar en el manual de referencia del programador cómo se hace esto en su computadora particular.

EJEMPLO 13.22. Consideremos la primera declaración de estructura presentada en el Ejemplo 13.20, que es

```
static struct {
                unsigned a : 5;
                unsigned b : 5;
                unsigned c : 5;
} v = {1, 2, 3};
```

Con algunas computadoras, el primer campo (`v.a`) ocupará los 5 bits más a la derecha (bits 0 a 4), el segundo campo (`v.b`) ocupará los 5 bits siguientes (bits del 5 al 9) y el último campo (`v.c`) ocupará del bit 10 al 14. El bit del extremo izquierdo (el bit 15, que es el bit más significativo) estará desocupado, como se muestra en la Figura 13.4(*a*).

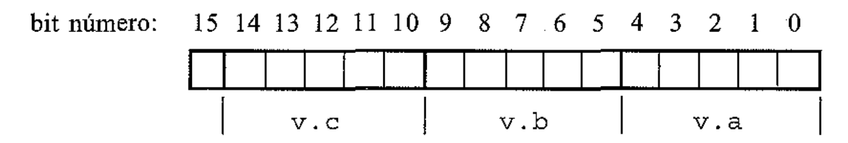


Figura 13.4(*a*). Campos de bits con ordenación de derecha a izquierda.

Sin embargo, con otras computadoras, el primer campo (`v.a`) ocupará los 5 bits de la izquierda (bits del `11` al `15`), el segundo campo (`v.b`) ocupará los bits del `6` al `10` y el último campo (`v.c`) ocupará los bits del `1` al `5`. El bit del extremo derecho (el menos significativo) quedará sin ocupar, como se muestra en la Figura 13.4(*b*). Por tanto, un programa escrito para un tipo de computadora puede producir resultados incorrectos cuando se ejecuta en otro tipo de computadora.

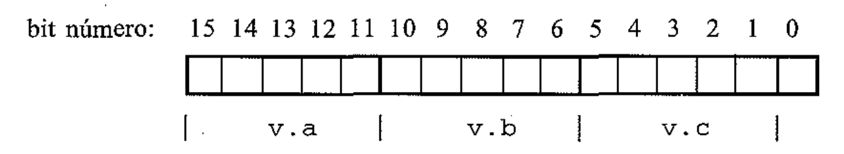


Figura 13.4(*b*). Campos de bits con ordenación de izquierda a derecha.

Los campos de bits son accedidos del mismo modo que otras estructuras, y pueden aparecer en expresiones aritméticas como cantidades enteras sin signo. Sin embargo, hay varias restricciones en su uso. En particular, no se permiten arrays de campos de bits; el operador dirección (`&`) no se puede aplicar a un campo de bits; un puntero no puede acceder a un campo de bits; y una función no puede devolver un campo de bits.

EJEMPLO 13.23. Compresión de datos (almacenamiento de nombres y fechas de nacimiento). Este ejemplo presenta un programa que almacena en un array nombres y fechas de nacimiento de varios estudiantes. La estrategia general será primero introducir el nombre y la fecha de nacimiento para cada estudiante. El programa mostrará para cada estudiante el nombre, el día de nacimiento (día de la semana en que nació el estudiante) y la fecha de nacimiento. Los días de nacimiento se determinarán usando el método descrito en el Ejemplo 10.28.

Cada fecha de nacimiento constará de tres números enteros: el mes, día y año de nacimiento. (El año se almacenará como un entero de tres dígitos que representan el número de años desde `1900`, como se describe en el Ejemplo 10.28. Así, el año `1999` se introduce como `1999` pero se almacena simplemente como `99`. Análogamente, el año `2010` se introducirá como `2010` y se almacenará como `110`.) Para ahorrar memoria, estos tres enteros se almacenarán en campos de bits dentro de una sola palabra de 16 bits, como sigue:

```
typedef struct {
                unsigned mes   : 4;
                unsigned dia   : 5;
                unsigned anio  : 7;
} fecha;
```

Observe que el mes se almacenará como un entero sin signo de 4 bits, cuyo valor puede variar de `0` a `15` (tenga en cuenta que $2^4 - 1 = 15$). Por supuesto, sólo interesan valores entre `1` y `12`. Análogamente,

el día se almacena como un entero sin signo de 5 bits. Su valor puede oscilar entre 0 y 31 (observe que $2^5 - 1 = 31$). Y el año se almacenará como un entero de 7 bits, que puede variar entre 0 y 127 (observe que $2^7 - 1 = 127$). Así, podremos acomodar fechas de nacimiento desde el año 1900 al año 2027.

Aquí tenemos el programa completo.

```
/* almacenar nombres y fechas de nacimiento de los estudiantes en un
   array, usando campos de bits para las fechas de nacimiento

   Cuando se termine, mostrar el nombre y la fecha de nacimiento de
   cada estudiante. Mostrar cada fecha de nacimiento como sigue: día
   de la semana, mes, día, año */

#include <stdio.h>
#include <string.h>

int convertir(int mm, int dd, int aa);/* prototipo de función */

main()
{
   int mm, dd, aa, cont = 0;
   int dia_semana;   /* día de la semana (0 -> Domingo, 1 -> Lu-
                        nes, etc.) */

   typedef struct {
                     unsigned mes   : 4;
                     unsigned dia   : 5;
                     unsigned anio  : 7;
   } fecha;

   struct {
            char nombre[30];
            fecha fechanacimiento;
   } estudiante[40];

   static char *diasemana[] = {    "Domingo", "Lunes", "Martes",
                              "Miércoles", "Jueves", "Viernes",
                              "Sábado"};

   static char *mes[] ={ "Enero", "Febrero", "Marzo", "Abril", "Mayo",
                         "Junio", "Julio", "Agosto", "Septiembre",
                         "Octubre", "Noviembre", "Diciembre"};
   /* mensaje de entrada */
   printf("Rutina de entrada de datos\nEscribir \'FIN\'#");
   printf("cuando se termine\n");
   printf("\nNombre: ");
   scanf(" %[^\n]", estudiante[cont].nombre);

   /* introducir los datos de todos los estudiantes */
   while (strcmp(estudiante[cont].nombre, "FIN") != 0) {
         printf("Fecha de nacimiento (mm dd aaaa): ");
         scanf("%d %d %d", &mm, &dd, &aa);
```

```
        /* asignar los datos enteros de la entrada a los campos de
           bits */
        estudiante[cont].fechanacimiento.mes  = mm;
        estudiante[cont].fechanacimiento.dia  = dd;
        estudiante[cont].fechanacimiento.anio = aa - 1900;

        printf("\nNombre: ");
        scanf(" %[^\n]", estudiante[++cont].nombre);
    }
    /* convertir las fechas de nacimiento y mostrar la salida para
       todos los estudiantes */
    cont = 0;
    while (strcmp(estudiante[cont].nombre, "FIN") != 0) {
        dia_semana = convertir(estudiante[cont].fechanacimiento.mes,
                               estudiante[cont].fechanacimiento.dia,
                               estudiante[cont].fechanacimiento.anio);
        printf("\n%s      ", estudiante[cont].nombre);
        printf("%s %s %d, %d\n", diasemana[dia_semana],
                      mes[estudiante[cont].fechanacimiento.mes-1],
                      estudiante[cont].fechanacimiento.dia,
                      estudiante[cont].fechanacimiento.anio + 1900);
        ++cont;
    }
}

int convertir(int mm, int dd, int aa) /* convertir una fecha en
                                          el día de la semana */
{
    long ndias;     /* número de días desde el comienzo de 1900 */
    long nciclos;   /* número de ciclos de 4 años después de 1900 */
    int nanios;     /* número de años después del último ciclo de 4 años */
    int dia;        /* día de la semana (0, 1, 2, 3, 4, 5 o 6) */

    /* conversiones numéricas */
    ndias = (long) (30.42 * (mm - 1)) + dd;   /* día aproximado del
                                                  año */

    if (mm == 2) ++ndias;                      /* ajuste para
                                                  febrero */
    if ((mm > 2) && (mm < 8)) --ndias;         /* ajuste para marzo-
                                                  julio */
    if ((aa % 4 == 0) && (mm > 2)) ++ndias;    /* ajuste para el año
                                                  bisiesto */

    nciclos = aa / 4;                          /* ciclos de 4 años
                                                  después de 1900 */
    ndias += nciclos * 1461;                   /* añadir días por ci-
                                                  clos de 4 años */
```

```
    nanios = aa % 4;       /* años después del último ciclo de 4 años */
    if (nanios > 0)        /* añadir días por años después del último
                              ciclo */
       ndias += 365 * nanios + 1;

    if (ndias > 59) --ndias; /* ajustar para 1900 (NO año bisiesto) */

    dia = ndias % 7;

    return(dia);
}
```

Dentro de este programa vemos que `estudiante` es un array de `40` estructuras. Cada elemento del array (cada estructura) consta de un array de caracteres de `30` elementos (`nombre`) que representa el nombre del estudiante, y otra estructura (`fechanacimiento`) que contiene la fecha de nacimiento del estudiante. Esta última estructura contiene tres campos de bits `fechanacimiento.mes`, `fechanacimiento.dia` y `fechanacimiento.anio` como miembros.

El programa tiene también dos arrays de cadenas de caracteres, cuyos elementos representan los días de la semana y los meses del año, respectivamente. Estos arrays se discuten en el Ejemplo 10.28. Además, el programa incluye la función `convertir`, que se usa para convertir cualquier fecha entre1-enero-1900 y 31-diciembre-2099 en el día de la semana equivalente (dado como entero). Esta función es idéntica a la descrita en el Ejemplo 10.28. (Dentro de `convertir`, la instrucción `aa -= 1900`, que estaba presente en el Ejemplo 10.28, está ahora ausente.)

La función `main` consiste esencialmente en dos bucles `while`. El primero se usa para introducir y almacenar los datos de cada estudiante. En cada pasada por el bucle se introducirán y almacenarán datos para un estudiante distinto. Este proceso continuará hasta que se detecte la palabra `"FIN"` para el nombre de un estudiante (en mayúsculas o minúsculas). Observe la forma en que se asignan valores a los campos de bits en este bucle.

El segundo bucle convierte la fecha de nacimiento en un día de la semana y lo muestra por pantalla, con el nombre y la fecha de nacimiento del estudiante. Los detalles de la conversión de la fecha y de la salida por pantalla de la información se dan en el Ejemplo 10.28 y no necesitan repetirse aquí. Observe cómo se acceden los campos de bits dentro de las llamadas a funciones.

El diálogo de entrada y la correspondiente salida se muestran a continuación. Como siempre, las respuestas del usuario están subrayadas.

```
Rutina de entrada de datos
Escribir 'FIN' cuando se termine

Nombre: Rob Smith
Fecha de nacimiento (mm dd aaaa): 7 20 1972

Nombre: Judy Thompson
Fecha de nacimiento (mm dd aaaa): 11 27 1983

Nombre: Jim Williams
Fecha de nacimiento (mm dd aaaa): 12 29 1998

Nombre: Mort Davis
Fecha de nacimiento (mm dd aaaa): 6 10 2010

Nombre: FIN
```

```
Rob Smith       Jueves Julio 20, 1972

Judy Thompson       Domingo Noviembre 27, 1983

Jim Williams      Martes Diciembre 29, 1998

Mort Davis      Jueves Junio 10, 2010
```

Unas observaciones adicionales antes de dejar este ejemplo. Primero, ha de destacarse que el ahorro de memoria usando campos de bits no es grande. El beneficio de esta técnica de compresión sería mayor si la dimensión del array de estudiantes se incrementara.

Segundo, la compresión de algunos datos adicionales se podría realizar almacenando ocho caracteres ASCII de 7 bits en 7 bytes de memoria, usando los operadores de desplazamiento a nivel de bits. Cada byte contendría un carácter completo más un bit del octavo carácter. Esto daría lugar a una reducción de un 12,5 por ciento en los requerimientos de memoria. Los detalles de esta técnica están más allá del ámbito de esta discusión, si bien los lectores interesados pueden querer experimentar esta técnica ellos solos. (Ver Problema 13.55 al final del capítulo.)

CUESTIONES DE REPASO

13.1. ¿Qué se entiende por programación a bajo nivel?

13.2. ¿Qué son los registros? En términos generales, ¿para qué se usan los registros?

13.3. ¿Cuál es el propósito del tipo de almacenamiento `register`? ¿Qué beneficios se obtienen del uso de este tipo de almacenamiento? ¿A qué tipo de variables puede asignarse este tipo de almacenamiento?

13.4. ¿Cuál es el ámbito de las variables registro?

13.5. Mencionar las reglas para el uso de variables registro.

13.6. ¿Por qué puede no ser tenida en cuenta una declaración `register`? Si la declaración `register` no se tiene en cuenta, ¿cómo se tratan estas variables?

13.7. ¿Cómo puede saber un programador si la declaración `register` se tiene en cuenta dentro de un programa?

13.8. ¿Qué se entiende por operaciones a nivel de bits?

13.9. ¿Cuál es el propósito del operador de complemento a uno? ¿A qué tipo de operandos se aplica? ¿A qué grupo de precedencia pertenece? ¿Cuál es su asociatividad?

13.10. Describir los tres operadores lógicos a nivel de bits. ¿Cuál es el propósito de cada uno?

13.11. ¿Qué tipo de operandos requiere cada uno de los operadores lógicos a nivel de bits?

13.12. Resumir los valores devueltos por cada una de las operaciones lógicas a nivel de bits. Considerar todos los posibles valores de los operandos en la respuesta.

13.13. Describir la precedencia y la asociatividad para cada uno de los operadores lógicos a nivel de bits.

13.14. ¿Qué es una operación de enmascaramiento? ¿Cuál es el propósito de cada operando? ¿Qué operando es la máscara y cómo se escoge?

13.15. Describir una operación de enmascaramiento en la cual se copia una parte del patrón de bits dado mientras que el resto de bits se ponen a cero. ¿Qué operación lógica a nivel de bits se usa para esta operación? ¿Cómo se escoge la máscara?

13.16. Describir una operación de enmascaramiento en la cual se copie una parte del patrón de bits dado mientras que el resto de los bits se ponen a 1. ¿Qué operación lógica a nivel de bits se usa para esta operación? ¿Cómo se define la máscara? Comparar la respuesta con la de la cuestión anterior.

13.17. Describir una operación de enmascaramiento en la cual se copie una parte del patrón de bits dado mientras que el resto de los bits se invierten. ¿Qué operación lógica a nivel de bits se usa para esta operación? ¿Cómo se define la máscara? Comparar la respuesta con la de las dos cuestiones anteriores.

13.18. ¿Por qué se usa a veces el operador de complemento a uno en una operación de enmascaramiento? ¿Bajo qué condiciones es deseable su uso?

13.19. ¿Cómo se puede conmutar un bit entre 0 y 1 de forma repetida? ¿Qué operación lógica a nivel de bits se utiliza para este propósito?

13.20. Describir los dos operadores de desplazamiento a nivel de bits. ¿Qué requerimientos deben satisfacer los operandos? ¿Cuál es el propósito de cada operando?

13.21. Describir la precedencia y la asociatividad de los operadores de desplazamiento a nivel de bits.

13.22. Cuando se desplazan bits hacia la derecha o hacia la izquierda, ¿qué pasa con los bits desplazados de su posición original fuera de la palabra?

13.23. Cuando se desplazan bits a la izquierda, ¿qué valor rellena las posiciones vacantes de la derecha de los bits desplazados?

13.24. Cuando se desplazan bits a la derecha, ¿qué valor rellena las posiciones vacantes de los bits de la izquierda desplazados? ¿Afecta a este valor el tipo de operando a desplazar? Explicarlo ampliamente. Comparar la respuesta con la de la Cuestión 13.23.

13.25. ¿Manejan todos los compiladores de C el desplazamiento a la derecha de la misma manera? Explicarlo ampliamente.

13.26. Listar los operadores de asignación a nivel de bits y describir su propósito.

13.27. Describir cada uno de los operandos en una operación de asignación a nivel de bits.

13.28. Describir la precedencia y la asociatividad para los operadores de asignación a nivel de bits.

13.29. ¿Qué son los campos de bits? ¿A qué tipo de estructura de datos pertenecen? ¿Cómo puede accederse a un campo de bits?

13.30. Mencionar las reglas para definir campos de bits.

13.31. ¿Qué tipo de datos debe asociarse con cada campo de bits?

13.32. ¿Qué pasa si un campo de bits sobrepasa una palabra de memoria de la computadora?

13.33. Dentro de la declaración de un campo de bits, ¿qué interpretación se le da a un campo de bits sin nombre? ¿Qué interpretación tiene un campo de bits de tamaño cero?

13.34. ¿En qué orden se colocan los campos de bits dentro de una palabra? ¿Este convenio es uniforme entre todos los compiladores?

13.35. ¿Qué restricciones se aplican al uso de campos de bits después de haber sido correctamente declarados?

PROBLEMAS

13.36. Declarar las variables u y v como variables enteras sin signo con el tipo de almacenamiento register.

13.37. Declarar las variables u, v, x e y como enteras con valores iniciales 1, 2, 3 y 4 respectivamente. Suponer que u y v son variables automáticas. Asignar el tipo de almacenamiento register a x e y.

13.38. Suponer que func es una función que acepta como argumento un puntero a una variable entera registro sin signo y devuelve un puntero a un entero sin signo. Escribir el esquema de la estructura de la rutina llamadora main y de func, ilustrando cómo se definen estas características.

13.39. Supongamos que a es una variable entera sin signo cuyo valor (hexadecimal) es 0xa2c3. Escribir el correspondiente patrón de bits para este valor. Evaluar a continuación cada una de las siguientes expresiones a nivel de bits, mostrando el patrón de bits resultante y su valor hexadecimal equivalente. Utilizar el valor original de a en cada expresión. Suponer que a se almacena en una palabra de 16 bits.

a) `~a`	*h*) `a >> 3`	*o*) `a & ~(0x3f06 << 8)`
b) `a & 0x3f06`	*i*) `a << 5`	*p*) `a ^ ~0x3f06 << 8`
c) `a ^ 0x3f06`	*j*) `a & ~a`	*q*) `(a ^ ~0x3f06) << 8`
d) `a | 0x3f06`	*k*) `a ^ ~a`	*r*) `a ^ ~(0x3f06 << 8)`
e) `a & ~0x3f06`	*l*) `a | ~a`	*s*) `a | ~0x3f06 << 8`
f) `a ^ ~0x3f06`	*m*) `a & ~0x3f06 << 8`	*t*) `(a | ~0x3f06) << 8`
g) `a | ~0x3f06`	*n*) `(a & ~0x3f06) << 8`	*u*) `a | ~(0x3f06 << 8)`

13.40. Reescribir cada una de las siguientes expresiones a nivel de bits en la forma de instrucciones de asignación a nivel de bits, donde el valor de cada expresión se asigna a la variable a.

a) Problema 13.39(*b*)
b) Problema 13.39(*c*)
c) Problema 13.39(*g*)
d) Problema 13.39(*h*)
e) Problema 13.39(*i*)
f) Problema 13.39(*k*)
g) Problema 13.39(*o*)

13.41. Definir una máscara y escribir la operación apropiada de enmascaramiento para cada una de las situaciones descritas a continuación.

a) Copiar los bits impares (bits 1, 3, 5, ..., 15) y colocar ceros en las posiciones pares (bits 0, 2, 4, ..., 14) de un entero sin signo de 16 bits representado por la variable v. Suponer que el bit 0 es el bit del extremo derecho.

b) Eliminar el bit más significativo (el bit del extremo izquierdo) de un carácter de 8 bits representado por la variable c. (Algunos procesadores de texto utilizan este bit para controlar el formato del texto dentro de un documento. Descartando este bit, por ejemplo poniéndolo a cero, puede transformarse un documento de un procesador de texto en un archivo de texto con caracteres ASCII ordinarios.)

c) Copiar los bits impares (bits 1, 3, 5, ..., 15) y colocar unos en las posiciones pares (bits 0, 2, 4, ..., 14) de un entero sin signo de 16 bits representado por la variable v. Suponer que el bit 0 es el bit del extremo derecho.

d) Conmutar (invertir) los valores de los bits 1 y 6 de una cantidad entera sin signo de 16 bits representada por la variable v, preservando el resto de los bits. Asignar el nuevo patrón de bits a v. Suponer que el bit 0 es el bit del extremo derecho.

13.42. *a*) Supongamos que v es una cantidad entera con signo de 16 bits cuyo valor hexadecimal es 0x369c. Evaluar cada una de las siguientes expresiones de desplazamiento. (Utilizar el valor original de v en cada expresión.)

i) v << 4
ii) v >> 4

b) Supongamos ahora que el valor de v se cambia a 0xc369. Evaluar cada una de las siguientes expresiones de desplazamiento y comparar los resultados con los obtenidos en la parte *a*). Explicar cualquier diferencia.

i) v << 4
ii) v >> 4

13.43. Describir la composición de cada una de las siguientes estructuras. Suponer una palabra entera de 16 bits.

a)
```
struct {
          unsigned u : 3;
          unsigned v : 1;
          unsigned w : 7;
          unsigned x : 5;
};
```

b)
```
static struct {
                 unsigned u : 3;
                 unsigned v : 1;
                 unsigned w : 7;
                 unsigned x : 5;
} a = {2, 1, 16, 8};
```

c)
```
struct {
          unsigned u : 7;
          unsigned v : 7;
          unsigned w : 7;
} a;
```

d)
```
struct {
          unsigned u : 7;
          unsigned   : 9;
          unsigned v : 7;
          unsigned   : 2;
          unsigned w : 7;
};
```

e)
```
struct {
          unsigned u : 7;
          unsigned   : 0;
          unsigned v : 7;
          unsigned   : 0;
          unsigned w : 7;
};
```

13.44. Escribir una declaración de estructura para cada una de las siguientes situaciones. Suponer una palabra entera de `16` bits.

a) Definir tres campos de bits llamados `a`, `b` y `c`, con tamaños de `6`, `4` y `6` bits, respectivamente.
b) Declarar una variable de tipo estructura `v` con la composición definida en la parte *a*). Asignar los valores iniciales `3`, `5` y `7`, respectivamente, a los tres campos de bits. ¿Son suficientemente grandes los campos para acomodar estos tres valores?
c) ¿Cuáles son los valores más grandes que se pueden asignar a cada uno de los campos de bits definidos en la parte *a*)?
d) Definir tres campos de bits llamados `a`, `b` y `c`, con tamaños de `8`, `6` y `5` bits, respectivamente. ¿Cómo se almacenan estos campos dentro de la memoria de la computadora?
e) Definir tres campos de bits llamados `a`, `b` y `c`, con tamaños de `8`, `6` y `5` bits, respectivamente. Separar `a` y `b` con dos bits vacantes.
f) Definir tres campos de bits llamados `a`, `b` y `c`, con tamaños de `8`, `6` y `5` bits, respectivamente. Forzar `b` al principio de la segunda palabra de almacenamiento. Separar `b` y `c` con dos bits vacantes.

PROBLEMAS DE PROGRAMACIÓN

13.45. Modificar el programa presentado en el Ejemplo 13.2 (cálculo repetido de una serie de números de Fibonacci) de modo que `f`, `f1` y `f2` sean punteros a enteros almacenados dentro de registros.

13.46. El Problema 6.69(*f*) describe un método para calcular números primos y sugiere escribir un programa para calcular los primeros *n* números primos, donde *n* es una cantidad especificada (por ejemplo, $n = 100$). Modificar este problema de modo que la lista de *n* números enteros se genere 30 000 veces. Mostrar la lista sólo una vez, después del paso a través del bucle.

Resolver el problema con y sin especificación de tipo de almacenamiento `register`. Comparar los tiempos de ejecución y los tamaños de los programas objeto compilados.

13.47. Otra forma de generar una lista de números primos es utilizar la famosa *criba de Eratóstenes*. Este método es como sigue:

a) Generar una lista ordenada de enteros entre 2 y *n*.
b) Para algunos enteros particulares *i* dentro de la lista, realizar las siguientes operaciones:
   i) Marcar el entero como primo (puede elegirse colocarlo en un array o escribirlo en un archivo).
   ii) Eliminar a continuación todos los enteros que son múltiplos de *i*.
c) Repetir la parte *b*) para cada valor sucesivo de *i* dentro de la lista, empezando con $i = 2$ y terminando con el último entero que quede.

Escribir un programa en C que use este método para determinar los primos dentro de una lista de números de 1 a *n*, donde *n* es una cantidad introducida por el usuario. Repetir los cálculos 30 000 veces mostrando la lista de números primos al final del último paso por el bucle.

Resolver el problema con y sin especificación de tipo de almacenamiento `register`. Comparar los tiempos de ejecución y los tamaños de los programas objeto compilados.

13.48. Escribir un programa en C que acepte un número hexadecimal como entrada y muestre un menú que permita realizar cualquiera de las siguientes operaciones:

a) Mostrar el hexadecimal equivalente del complemento a uno.
b) Realizar una operación de enmascaramiento y mostrar el hexadecimal equivalente del resultado.
c) Realizar una operación de desplazamiento de bits y mostrar el hexadecimal equivalente del resultado.
d) Salir.

Si se selecciona la operación de enmascaramiento, pedir al usuario el tipo de operación (*y a nivel de bits*, *o exclusiva a nivel de bits*, u *o a nivel de bits*) y luego un valor (hexadecimal) para la máscara. Si se selecciona la operación de desplazamiento, pedir al usuario el tipo de desplazamiento (derecha o izquierda) y el número de bits.

Comprobar el programa con varios valores diferentes de su propia elección.

13.49. Modificar el programa escrito para el Problema 13.48 de modo que se muestren los patrones binarios de bits además de los valores hexadecimales. Usar una función separada para mostrar los patrones binarios, tomando como modelo el programa mostrado en el Ejemplo 13.16.

13.50. Modificar el programa escrito para el Problema 13.49 de modo que la cantidad de entrada pueda ser una constante decimal, hexadecimal u octal. Empezar mostrando un menú para permitir al usuario elegir el tipo de número (el sistema de numeración deseado) antes de introducir el valor real. Después mostrar el valor en los otros dos sistemas de numeración y en términos de su correspondiente patrón de bits.

Una vez que la cantidad ha sido introducida y mostrada, generar el menú principal pidiendo el tipo de operación deseada, como se describe en el Problema 13.48. Si se selecciona una operación de enmascaramiento, introducir la máscara como una constante hexadecimal u octal. Mostrar el resultado de cada operación en decimal, hexadecimal, octal y binario.

13.51. Escribir un programa en C que ilustre la equivalencia entre:

a) Desplazar un número binario n posiciones a la izquierda y multiplicarlo por 2^n.
b) Desplazar un número binario n bits a la derecha y dividirlo por 2^n (o equivalentemente multiplicarlo por 2^{-n}).

Elegir el número binario inicial con cuidado de modo que no se pierdan bits como resultado de las operaciones de desplazamiento. (Para el desplazamiento a la izquierda, elegir un número relativamente pequeño de modo que tenga varios ceros en las posiciones de más a la izquierda. Para el desplazamiento a la derecha, elegir un número relativamente grande, con ceros en las posiciones más a la derecha.)

13.52. Escribir un programa completo en C que codifique y decodifique el contenido de un archivo de texto (archivo orientado a caracteres) mediante el reemplazamiento de cada carácter con su complemento a uno. Observe que el complemento a uno del complemento a uno de un carácter es el carácter original. Por tanto, el proceso de obtener el complemento a uno puede usarse tanto para codificar el texto original como para decodificar el texto codificado.

Incluir las siguientes características en el programa:

a) Introducir el contenido de un archivo ordinario de texto desde el teclado.
b) Grabar el archivo de texto actual en su estado presente (codificado o decodificado).
c) Recuperar el texto que ha sido grabado (codificado o decodificado).
d) Codificar o decodificar el archivo de texto actual (cambiar el estado actual obteniendo el complemento a uno de cada uno de los caracteres).
e) Mostrar el archivo de texto actual en su estado presente (codificado o decodificado).

Generar un menú que permita al usuario seleccionar cualquiera de estas características según desee.

13.53. Modificar el programa escrito para el Problema 13.52 de modo que la codificación y decodificación se realicen usando la *o exclusiva a nivel de bits* o una operación de enmascaramiento en vez de la operación de complemento a uno. Permitir al usuario introducir una *clave* (una máscara que será el segundo operando en la operación *o exclusiva*). Como la operación *o exclusiva* produce una operación de conmutación, se puede usar para codificar el texto original o para decodificar el texto codificado. Se debe usar la misma clave para codificar y decodificar.

13.54. Modificar el programa de compresión de datos mostrado en el Ejemplo 13.23 de modo que muestre la edad de cada estudiante (en años) además de la salida que genera normalmente. Añadir las siguientes capacidades como opciones separadas:

a) Mostrar la edad de un estudiante cuyo nombre y fecha de nacimiento se introduzcan como entrada.
b) Mostrar los nombres de los estudiantes cuya edad sea la especificada por el usuario.
c) Mostrar los nombres de los estudiantes cuya edad sea la especificada por el usuario o menor.
d) Mostrar los nombres de los estudiantes cuya edad sea la especificada por el usuario o mayor.

Generar un menú que permita al usuario seleccionar cualquiera de estas opciones según desee.

13.55. Modificar el programa presentado en el Ejemplo 10.8 (análisis de una línea de texto) de modo que los 80 caracteres de la línea de texto se almacenen en un array de caracteres de 70 bytes. (Suponer caracteres ASCII de 7 bits.) Para hacer esto, usar los operadores de desplazamiento a nivel de bits de manera que un grupo de ocho caracteres se almacene en siete elementos consecutivos del array (siete bytes). Cada elemento del array contendrá un carácter completo, más un bit del octavo carácter.

Incluir la posibilidad de mostrar los contenidos del array de 70 bytes (usando constantes hexadecimales) en la forma comprimida y en la forma no comprimida equivalente.

Usar el programa para analizar la siguiente línea de texto:

```
Las computadoras personales con más de 1024 KB son ahora bastante
comunes.
```

(Observe que esta línea de texto, incluyendo la puntuación y los espacios en blanco entre las palabras, contiene un total de 74 caracteres.) Examinar la salida hexadecimal, así como los resultados del análisis para verificar que el programa se ejecuta correctamente.

CAPÍTULO 14

Características adicionales de C

En este último capítulo consideramos algunas características no tratadas de C y presentamos información sobre otras características previamente tratadas. Comenzamos con una discusión de las enumeraciones, un tipo de datos que define un conjunto de identificadores de tipo entero que pueden ser asignados a las correspondientes variables de enumeración. Las variables de enumeración son útiles en programas que requieren indicadores para identificar condiciones lógicas internas.

A continuación tratamos los argumentos de la línea de órdenes, los cuales permiten que los parámetros sean transferidos al programa cuando el programa objeto compilado se ejecuta desde el sistema operativo. Por ejemplo, los nombres de archivos pueden ser transferidos fácilmente al programa de esta manera.

Se presenta una discusión de la biblioteca de funciones de C, en la cual son consideradas, desde una amplia perspectiva, las funciones soportadas por la mayoría de los compiladores comerciales de C. Esto es seguido por la discusión de las macros, que proporcionan una alternativa al uso de funciones de biblioteca. El uso de macros puede ser más aconsejable que el uso de funciones de biblioteca en determinadas situaciones. El capítulo concluye con una discusión sobre el preprocesador de C, que es un conjunto especial de órdenes que se ejecutan al principio del proceso de compilación.

14.1. ENUMERACIONES

Una *enumeración* es un tipo de datos, similar a la estructura o a la unión. Sus miembros son constantes que están escritas como identificadores, pero que toman valores enteros con signo. Estas constantes representan valores que pueden ser asignados a variables de enumeración.

En términos generales, una enumeración puede ser definida como

```
enum marca {miembro 1, miembro 2, . . ., miembro m};
```

donde `enum` es la palabra reservada requerida, *`marca`* el nombre que identifica la enumeración con estos componentes y *`miembro 1`*, *`miembro 2`*, . . ., *`miembro m`* representan los identificadores individuales que pueden ser asignados a variables de este tipo (ver más adelante). Los nombres de miembros deben ser todos diferentes y deben ser distintos de otros identificadores cuyo ámbito sea el mismo que el de la enumeración.

Una vez que la enumeración ha sido definida, las correspondientes variables de enumeración pueden ser definidas como

```
tipo-almacenamiento enum  marca  variable 1,
                              variable 2, . . ., variable n;
```

donde *tipo-almacenamiento* es un especificador opcional de tipo de almacenamiento, enum la palabra reservada requerida, *marca* el nombre que aparece en la definición de la enumeración y *variable 1*, *variable 2*, . . ., *variable n* variables de enumeración del tipo *marca*.

La definición de enumeración puede ser combinada con la declaración de variables de este tipo, como se muestra a continuación.

```
tipo-almacenamiento enum  marca  {miembro 1,
                   miembro 2, . . ., miembro m} variable 1,
                              variable 2, . . ., variable n;
```

La *marca* es opcional en esta situación.

EJEMPLO 14.1. Un programa en C contiene las siguientes declaraciones:

```
enum colores {negro, azul, cian, verde, magenta, rojo, blanco, amarillo};

colores primerplano, fondo;
```

La primera línea define una enumeración llamada colores (la marca es colores). La enumeración consta de ocho constantes cuyos nombres son negro, azul, cian, verde, magenta, rojo, blanco y amarillo.

La segunda línea declara las variables primerplano y fondo como variables enumeradas del tipo colores. Así, a cada variable se le puede asignar cualquiera de las constantes negro, azul, cian, ..., amarillo.

Las dos declaraciones pueden ser combinadas si se desea, dando como resultado:

```
enum colores {negro, azul, cian, verde, magenta, rojo, blanco, amarillo}
                                                  primerplano, fondo;
```

o, sin la marca, simplemente

```
enum {negro, azul, cian, verde, magenta, rojo, blanco, amarillo}
                                                  primerplano, fondo;
```

Las constantes de enumeración tienen automáticamente asignados valores enteros, empezando por 0 para la primera constante e incrementándose en 1 para las sucesivas. De esta manera, *miembro 1* tendrá automáticamente asignado el valor 0, *miembro 2* tendrá asignado 1, y así sucesivamente.

EJEMPLO 14.2. Consideremos la enumeración definida en el Ejemplo 14.1, esto es,

```
enum colores {negro, azul, cian, verde, magenta, rojo, blanco, amarillo};
```

Las constantes de enumeración representarán ahora los siguientes valores enteros:

```
negro         0
azul          1
cian          2
verde         3
magenta       4
rojo          5
blanco        6
amarillo      7
```

Estas asignaciones automáticas pueden ser modificadas dentro de la definición de la enumeración. Esto es, a algunas constantes se les pueden asignar valores enteros diferentes de los valores por omisión. Para hacer esto, cada constante (cada miembro) a la que es asignado un valor explícito se expresa como una expresión de asignación ordinaria; por ejemplo, *miembro = int*, donde *int* representa una cantidad entera con signo. A las constantes que no tengan asignados valores explícitos se les asignarán automáticamente los valores incrementados sucesivamente en 1 desde la última asignación explícita. Esto puede hacer que dos o más constantes de enumeración tengan el mismo valor entero.

EJEMPLO 14.3. Aquí tenemos una variación de la enumeración definida en los Ejemplos 14.1 y 14.2.

```
enum colores {negro = -1,azul, cian, verde, magenta, rojo = 2,
                         blanco, amarillo};
```

Las constantes de enumeración representarán ahora los siguientes valores enteros:

```
negro      -1
azul        0
cian        1
verde       2
magenta     3
rojo        2
blanco      3
amarillo    4
```

Las constantes negro y rojo tienen ahora asignados explícitamente los valores -1 y 2, respectivamente. Las otras constantes toman automáticamente valores incrementados sucesivamente en 1 desde la última asignación explícita. Así, azul, cian, verde y magenta tienen asignados los valores 0, 1, 2 y 3, respectivamente. Análogamente, blanco y amarillo tienen asignados los valores 3 y 4. Observe que ahora tenemos asignaciones duplicadas; por ejemplo, verde y rojo representan ambos 2, mientras que magenta y blanco representan 3.

Las variables enumeración pueden ser tratadas de la misma manera que otras variables enteras. Esto es, pueden asignárseles nuevos valores, compararse, etc. Sin embargo, debe quedar claro que las variables de enumeración son generalmente usadas internamente para indicar condiciones que pueden surgir dentro de un programa. Por tanto, hay ciertas restricciones asociadas con su uso. En particular, *una constante de enumeración no puede ser leída por la computadora y asignada a una variable de enumeración.* (Es posible leer un *entero* y asignarlo a una variable de enumeración, aunque generalmente no se hace.) Además, *sólo el valor entero de una variable de enumeración puede ser escrito por la computadora.*

EJEMPLO 14.4. Consideremos una vez más las declaraciones presentadas en el Ejemplo 14.1, es decir,

```
enum colores {negro, azul, cian, verde, magenta, rojo, blanco, amarillo};
colores primerplano, fondo;
```

A continuación se muestran varias instrucciones que usan las variables de enumeración `primerplano` y `fondo`.

```
primerplano = blanco;

fondo = azul;

if (fondo == azul)
   primerplano = amarillo;
else
   primerplano = blanco;

if (primerplano == fondo)
   fondo = (enum colores) (++fondo % 8);

switch (fondo) {

case negro:
            primerplano = blanco;
            break;

case azul:
            azul:
            cian:
            verde:
            magenta:
            rojo:
            primerplano = amarillo;
            break;

case blanco:
            primerplano = negro;
            break;

case amarillo:
            primerplano = azul;
            break;

case default:
            printf("ERROR EN LA SELECCION DEL COLOR DE FONDO\n");
}
```

A menudo se puede incrementar la claridad lógica de un programa usando variables de enumeración. Las variables de enumeración son particularmente útiles como indicadores, para indicar varias opciones para realizar un cálculo, o para identificar condiciones que puedan haber surgido como resultado de cálculos internos anteriores. Desde esta perspectiva se aconseja el

uso de variables de enumeración dentro de un programa complejo. No obstante, debe quedar claro que se pueden usar variables enteras en vez de variables de enumeración. Por tanto, las variables de enumeración no proporcionan fundamentalmente nuevas capacidades.

EJEMPLO 14.5. Elevación de un número a una potencia. En el Ejemplo 11.37 vimos un programa en C para evaluar la fórmula $y = x^n$, donde x e y son valores en coma flotante y n un exponente entero o en coma flotante. Dicho programa hacía uso de las siguientes estructuras de datos:

```
typedef union {
    float fexp;              /* exponente en coma flotante */
    int nexp;                /* exponente entero */
} nvalor;

typedef struct {
    float x;                 /* valor para elevar a la potencia */
    char indicador;          /* 'f' si el exponente es en coma flotante
                                'e' si el exponente es entero */
    nvalor exp;              /* unión que contiene el exponente */
} valores;
```

Observe que la unión contiene el valor del exponente, que puede ser una cantidad entera o en coma flotante. La estructura incluye el valor de x, un indicador (un carácter que indica la naturaleza del exponente) y la unión que contiene el exponente.

Ahora presentamos otra versión, en la cual el carácter indicador es reemplazado por una variable de enumeración. La estructura de datos queda modificada como sigue:

```
typedef enum {exp_float, exp_int} tipo_exp;

typedef union {
    float fexp;              /* exponente en coma flotante */
    int nexp;                /* exponente entero */
} nvalor;

typedef struct {
    float x;                 /* valor para elevar a la potencia */
    tipo_exp indicador;      /* indicador del tipo de exponente */
    nvalor exp;              /* unión que contiene el exponente */
} valores;
```

Observe que `indicador`, que es un miembro de la estructura del tipo `valores`, es ahora una variable de enumeración del tipo `tipo_exp`. Esta variable puede tomar el valor de `exp_float` o `exp_int`, indicando un exponente en coma flotante o entero respectivamente.

Los cálculos serán realizados de modo distinto dependiendo del tipo de exponente. En particular, la potenciación se efectuará mediante multiplicaciones en el caso de exponente entero y utilizando logaritmos en el caso de exponente en coma flotante.

Aquí está la versión modificada del programa.

```
/* programa para elevar un número a una potencia */

#include <stdio.h>
#include <math.h>
```

```
typedef enum {exp_float, exp_int} tipo_exp;

typedef union {
    float fexp;             /* exponente en coma flotante */
    int nexp;               /* exponente entero */
} nvalor;

typedef struct {
    float x;                /* valor para elevar a la potencia */
    tipo_exp indicador;     /* indicador del tipo de exponente */
    nvalor exp;             /* unión que contiene el exponente */
} valores;

float potencia(valores a); /* prototipo de función */

main()
{
    valores a;              /* estructura que contiene x, indicador,
                               fexp/nexp */
    int i;
    float n, y;

    /* introducción de datos de entrada */
    printf("y = x^n\n\nIntroducir un valor para x: ");
    scanf("%f", &a.x);
    printf("Introducir un valor para n: ");
    scanf("%f", &n);

    /* determinar el tipo de exponente */
    i = (int) n;
    a.indicador = (i == n) ? exp_int : exp_float;
    if (a.indicador == exp_int)
       a.exp.nexp = i;
    else
       a.exp.fexp = n;

    /* elevar x a la potencia adecuada y mostrar el resultado */
    if (a.indicador == exp_float && a.x <= 0.0) {
       printf("\nERROR - No se puede elevar un número no positivo a ");
       printf("una potencia en coma flotante");
    }
    else {
       y = potencia(a);
       printf("\ny = %.4f", y);
    }
}

float potencia(valores a)   /* realiza la potenciación */
{
    int i;
    float y = a.x;
```

```
    if (a.indicador == exp_int) { /* exponente entero */
        if (a.exp.nexp == 0)
           y = 1.0;                 /* exponente cero */
        else {
           for (i = 1; i < abs(a.exp.nexp); ++i)
               y *= a.x;
           if (a.exp.nexp < 0)
               y = 1./y;            /* exponente entero no negativo */
        }
    }
    else                            /* exponente en coma flotante */
        y = exp(a.exp.fexp * log(a.x));

    return(y);
}
```

Cuando se ejecuta este programa, se comporta de la misma manera que la versión anterior. Esto se puede verificar ejecutando el programa con los datos de entrada del Ejemplo 11.37.

Esta versión del programa no representa una gran mejora sobre la versión anterior. Sin embargo, la ventaja del uso de variables de enumeración es mayor en programas que incluyen opciones más complicadas.

Una variable de enumeración puede ser inicializada en general de la misma manera que otras variables en C. La inicialización puede ser realizada asignando una constante de enumeración o un valor entero a la variable. Sin embargo, a la variable se le asignará una constante de enumeración, como se ilustra a continuación (ver también el Ejemplo 14.13).

EJEMPLO 14.6. Un programa en C contiene las siguientes declaraciones:

```
enum  colores {negro, azul, cian, verde, magenta, rojo, blanco, amarillo};

colores  primerplano = amarillo, fondo = rojo;
```

Así, a las variables de enumeración `primerplano` y `fondo` se le asignan los valores iniciales `amarillo` y `rojo`, respectivamente. Estas asignaciones de inicialización son equivalentes a escribir

```
primerplano = 7;

fondo = 5;
```

Sin embargo, generalmente a las variables de enumeración se le asignan constantes de enumeración en vez de números enteros equivalentes.

14.2. PARÁMETROS DE LA LÍNEA DE ÓRDENES

¿Se ha estado usted preguntando para qué sirven los paréntesis vacíos en la primera línea de la función `main`, esto es, `main()`? Estos paréntesis pueden contener argumentos especiales que permiten pasarle parámetros a `main` desde el sistema operativo. La mayoría de las versiones

de C permiten dos argumentos, que se llaman tradicionalmente `argc` y `argv`, respectivamente. El primero de ellos, `argc`, debe ser una variable entera, mientras que el segundo, `argv`, es una formación de punteros a carácter, es decir, una formación de cadenas de caracteres. Cada cadena en esta formación representará un parámetro que es pasado a `main`. El valor de `argc` indicará el número de parámetros pasados.

EJEMPLO 14.7. El siguiente esquema indica cómo están definidos los argumentos `argc` y `argv` dentro de `main`.

```
void main (int argc, char *argv[])
{
   . . . . .
}
```

La primera línea se puede escribir sin la palabra reservada `void`, esto es:

```
main (int argc, char *argv[])
```

Un programa se ejecuta normalmente especificando el nombre del programa en un entorno guiado por menús, como se explicó en la sección 5.4. Algunos compiladores también permiten que el programa sea ejecutado especificando el nombre del programa (en realidad el nombre del archivo que contiene el programa objeto compilado) en el *nivel del sistema operativo*. El nombre del programa es interpretado como una orden del sistema operativo. De aquí que la línea en la que aparecen se denomine generalmente *línea de órdenes* o *comandos*.

Para pasar uno o más parámetros al programa cuando se ejecuta desde el sistema operativo, los parámetros deben seguir al nombre del programa en la línea de órdenes, por ejemplo,

nombre-programa parámetro 1 parámetro 2 . . . parámetro n

Los datos individuales deben ir separados por espacios en blanco o tabuladores. Algunos sistemas operativos permiten incluir espacios en blanco dentro de un parámetro, siempre que el parámetro esté entre comillas.

El nombre del programa será almacenado como el primer elemento en `argv`, seguido por cada uno de los parámetros. Por tanto, si el nombre del programa es seguido por *n* parámetros, habrá (*n* + 1) entradas en `argv`, dentro del rango que va desde `argv[0]` hasta `argv[n]`. Además, `argc` tendrá automáticamente asignado el valor (*n* + 1). Tenga presente que el valor para `argc` no viene dado explícitamente desde la línea de órdenes.

EJEMPLO 14.8. Consideremos el siguiente programa sencillo en C, que se ejecuta desde una línea de órdenes.

```
#include <stdio.h>

main(int argc, char *argv[])
{
   int cont;
```

```
    printf("argc = %d\n", argc);

    for (cont = 0; cont < argc; ++cont)
        printf("argv[%d] = %s\n", cont, argv[cont]);
}
```

Este programa permitirá introducir un número no especificado de parámetros desde la línea de órdenes. Cuando el programa es ejecutado, el valor actual de `argc` y los elementos de `argv` serán mostrados en líneas de salida separadas.

Supongamos, por ejemplo, que el nombre del programa es `muestra` y la línea de órdenes que inicia la ejecución del programa es

```
muestra  rojo  blanco  azul
```

Entonces la ejecución del programa tendrá como resultado la siguiente salida:

```
argc = 4
argv[0] = muestra.exe
argv[1] = rojo
argv[2] = blanco
argv[3] = azul
```

La salida nos indica que han sido introducidos cuatro elementos desde la línea de órdenes. El primero es el nombre del programa, `muestra.exe`, seguido por tres parámetros, `rojo`, `blanco` y `azul`. Cada uno es un elemento en la formación `argv`. (Observe que `muestra.exe` es el nombre del archivo objeto resultado de la compilación del código fuente `muestra.c`.)

Análogamente, si la línea de órdenes es

```
muestra  rojo  "blanco  azul"
```

la salida resultante será

```
argc = 3
argv[0] = muestra.exe
argv[1] = rojo
argv[2] = blanco azul
```

En este caso la cadena de caracteres `"blanco  azul"` será interpretada como un parámetro simple, debido a las comillas.

Una vez que los parámetros han sido introducidos, pueden ser utilizados dentro del programa como se desee. Una aplicación común es especificar nombres de archivos de datos como parámetros de la línea de órdenes. Esta técnica está ilustrada a continuación.

EJEMPLO 14.9. Lectura de un archivo de datos. Aquí tenemos una variación del programa mostrado en el Ejemplo 12.4, que lee una línea de texto desde un archivo de datos carácter a carácter y muestra el

texto en la pantalla. En su forma original, el programa leía el texto desde un archivo de datos llamado `muestra.dat`, donde el nombre del archivo estaba incluido dentro del programa. Sin embargo, ahora el nombre del archivo de datos es introducido como un parámetro de la línea de órdenes. Así el programa es aplicable para *cualquier* archivo de datos; ya no está restringido a `muestra.dat`.

Aquí tenemos el programa entero.

```
/* leer una línea de texto de un archivo de datos y mostrarla en
   pantalla */

#include <stdio.h>

#define  NULL  0

main(int argc, char *argv[])
{
   FILE *fpt; /* define un puntero a la estructura predefinida FILE */

   char c;

   /* abrir el archivo de datos solo para lectura */
   if ((fpt = fopen(argv[1], "r")) == NULL)
      printf("\nERROR - No se puede abrir el archivo designado \n");

   else   /* leer y mostrar cada carácter desde el archivo de datos */
      do
         putchar(c = getc(fpt));
      while (c != '\n');

   /* cerrar el archivo de datos */
   fclose(fpt);
}
```

Observe que la función `main` incluye ahora los argumentos formales `argc` y `argv`, definidos de la manera descrita anteriormente. También, la función `fopen` está ahora escrita como

```
fopen(argv[1], "r")
```

en vez de

```
fopen("muestra.dat", "r")
```

como en la anterior versión de este programa.

Supongamos ahora que el nombre del programa es `leerarchivo`. Para ejecutar este programa y leer el archivo de datos `muestra.dat`, la línea de órdenes se escribiría

```
leerarchivo  muestra.dat
```

El programa se comportará de la misma manera que la versión anterior mostrada en el Ejemplo 12.4.

13.4. MÁS SOBRE FUNCIONES DE BIBLIOTECA

Hasta ahora hemos aprendido que las funciones de biblioteca de C son extensas, tanto en número como en propósito. Hemos tenido evidencia de esto en los ejemplos de programas presentados previamente en este libro y en la lista de funciones de biblioteca comúnmente usadas dadas en el Apéndice H. A lo largo de este libro hemos usado esas funciones libremente, en cualquier sitio donde ha sido necesario.

Sin embargo, debe quedar claro que todas las funciones de biblioteca presentadas en este libro entran dentro de algunas categorías básicas. En particular, han facilitado operaciones de entrada/salida, evaluación de funciones matemáticas, conversiones de datos, clasificación y conversión de caracteres, manipulación de cadenas de caracteres, gestión de memoria dinámica y algunas operaciones variadas relacionadas con el reloj.

La mayoría de los compiladores comerciales de C incluyen muchas funciones de biblioteca adicionales. Algunas de esas funciones entran dentro de las categorías descritas antes, mientras que otras corresponden a categorías que todavía no han sido descritas en este libro. Por ejemplo, la mayoría de los compiladores incluyen funciones de biblioteca que pueden manipular áreas de buffer (bloques de memoria en que las formaciones de caracteres pueden ser almacenados temporalmente), que facilitan el manejo y gestión de archivos y que proporcionan la capacidad de realizar búsquedas y ordenaciones. Además pueden existir funciones de biblioteca que permitan el acceso a ciertas órdenes del sistema operativo y al hardware interno de la computadora (especialmente instrucciones contenidas en la memoria de sólo lectura de la computadora). Finalmente, algunos compiladores incluyen funciones de biblioteca para aplicaciones más específicas, como el control de procesos y gráficos.

Estas funciones de biblioteca simplifican, en un número importante de áreas, la escritura de programas de fácil comprensión en C. Por ejemplo, C se usa tanto para escribir sistemas operativos como para aplicaciones ofimáticas tales como procesadores de texto, hojas de cálculo y programas de gestión de bases de datos. El famoso sistema operativo UNIX está escrito en C. También lo están muchos programas ofimáticos.

Las funciones de control de procesos permiten aplicaciones en las que varios programas son ejecutados a la vez de manera jerárquica. Análogamente, las funciones gráficas simplifican la escritura de variadas aplicaciones gráficas, tales como programas de «pintura» y aplicaciones de diseño asistido por computadora (CAD). El uso de C para otros tipos de aplicaciones comerciales parece estar creciendo rápidamente.

Una discusión detallada de estas aplicaciones completas de programación está más allá del ámbito del presente texto. Sin embargo, el lector debe tener en cuenta que es práctico escribir tales aplicaciones en C, principalmente debido a la disponibilidad de la amplia biblioteca de C. Los lectores que quieran seguir profundizando en estos temas deben familiarizarse con la biblioteca de funciones suministrada con su compilador de C particular.

14.4. MACROS

Ya hemos visto que la instrucción `#define` puede ser usada para definir constantes dentro de un programa en C. Al principio del proceso de compilación todas las constantes simbólicas son reemplazadas por su texto equivalente (ver sección 2.9). Así, las constantes simbólicas proporcionan una forma de notación abreviada que puede simplificar la organización de un programa.

Sin embargo, la instrucción `#define` puede ser usada para definir algo más que constantes simbólicas. En particular, puede ser usada para definir *macros*; es decir, identificadores simples que son equivalentes a expresiones, a instrucciones completas o a grupos de instrucciones. En este sentido las macros se parecen a las funciones. No obstante, son definidas y tratadas de forma diferente que las funciones durante el proceso de compilación.

EJEMPLO 14.10. Consideremos el programa sencillo en C mostrado a continuación.

```
#include <stdio.h>

#define  area  longitud * anchura

main()
{
   int longitud, anchura;

   printf("longitud = ");
   scanf("%d", &longitud);
   printf("anchura = ");
   scanf("%d", &anchura);

   printf("\narea = %d", area);
}
```

Este programa contiene la macro `area`, que representa la expresión `longitud * anchura`. Cuando el programa es compilado, la expresión `longitud * anchura` reemplazará al identificador `area` dentro de la instrucción `printf`, de tal forma que la instrucción `printf` quedará

```
printf("area = %d", longitud * anchura);
```

Observe que la cadena `"\narea = %d"` no se ve afectada por la instrucción `#define` (ver sección 2.9).

Cuando se ejecuta el programa, los valores para `longitud` y `anchura` son introducidos interactivamente desde el teclado y el valor correspondiente para `area` es mostrado por pantalla. A continuación se presenta una sesión interactiva típica. Las respuestas del usuario están subrayadas como de costumbre.

```
longitud = 3
anchura = 4

area = 12
```

Las definiciones de macros están normalmente colocadas al principio de un archivo, antes de la definición de la primera función. El ámbito de una definición de macro va desde el punto de definición hasta el final del archivo donde ha sido definida. Sin embargo, una macro definida en un archivo no es reconocida dentro de otro archivo.

Podemos definir macros con varias líneas colocando una barra invertida (\) al final de cada línea, excepto en la última. Esta característica permite que una sola macro (un identificador simple) represente una instrucción compuesta.

EJEMPLO 14.11. He aquí otro programa sencillo en C que contiene una macro.

```
#include <stdio.h>

#define bucle for (lineas = 1; lineas <= n; lineas++) {        \
                   for (cont = 1; cont <= n - lineas; cont++)   \
                        putchar(' ');                           \
                   for (cont = 1; cont <= 2 * lineas - 1; cont++)\
                      putchar('*');                             \
                   printf("\n");                                \
              }
main()
{
   int cont, lineas, n;

   printf("número de líneas = ");
   scanf("%d", &n);
   printf("\n");

   bucle;
}
```

Este programa contiene una macro de varias líneas, que representa a una instrucción compuesta. La instrucción compuesta consta de varios bucles `for` anidados. Observe la barra invertida (\) incluida al final de las líneas, excepto en la última.

Cuando el programa es compilado, la referencia a la macro es reemplazada por las instrucciones contenidas dentro de la definición de la macro. Así, el programa mostrado anteriormente se convierte en

```
#include <stdio.h>

main()
{
   int cont, lineas, n;

   printf("número de líneas = ");
   scanf("%d", &n);
   printf("\n");

   for (lineas = 1; lineas <= n; lineas++) {
        for (cont = 1; cont <= n - lineas; cont++)
             putchar(' ');
        for (cont = 1; cont <= 2 * lineas - 1; cont++)
             putchar('*');
        printf("\n");
   }
}
```

Cuando el programa es ejecutado, muestra un triángulo de asteriscos, con un número de líneas determinado por el valor dado por el usuario (el valor de *n*). A continuación se muestra el resultado de una ejecución típica. De nuevo la respuesta del usuario está subrayada.

```
número de líneas: 6

     *
    ***
   *****
  *******
 *********
***********
```

Una definición de macro puede incluir argumentos, que están encerrados entre paréntesis. El paréntesis izquierdo debe aparecer inmediatamente detrás del nombre de la macro; es decir, no pueden existir espacios entre el nombre de la macro y el paréntesis izquierdo. Cuando una macro es definida de esta manera, su aspecto dentro del programa es similar a la llamada a una función.

EJEMPLO 14.12. He aquí una variación del programa mostrado en el Ejemplo 14.11.

```
#include <stdio.h>

#define  bucle(n)  for (lineas = 1; lineas <= n; lineas++) {       \
                       for (cont = 1; cont <= n - lineas; cont++)   \
                           putchar(' ');                            \
                       for (cont = 1; cont <= 2 * lineas - 1; cont++)\
                           putchar('*');                            \
                       printf("\n");                                \
                   }
main()
{
   int cont, lineas, n;

   printf("número de líneas = ");
   scanf("%d", &n);
   printf("\n");

   bucle(n);
}
```

El programa pasa ahora el valor de *n* a la macro, como si fuera un argumento real en la llamada de una función.

Este programa se comporta de la misma manera que el programa mostrado en el Ejemplo 14.11 cuando es ejecutado.

A veces se usan macros en lugar de funciones dentro de un programa. El uso de una macro en lugar de una función elimina el retraso asociado con la llamada a la función. Si el programa contiene muchas llamadas a funciones repetidas, el tiempo ahorrado por el uso de macros puede ser significativo.

Por otra parte, la sustitución de la macro se realizará en todas las referencias a la macro que aparezcan dentro de un programa. Así, un programa que contenga varias referencias a la misma macro puede volverse excesivamente largo. Por tanto, tenemos la disyuntiva entre la velocidad

de ejecución y el tamaño del programa objeto compilado. El uso de una macro es más ventajoso en aplicaciones donde hay relativamente pocas llamadas a funciones pero la función es llamada repetidamente (por ejemplo, una función llamada dentro de un bucle).

EJEMPLO 14.13. Valor futuro de depósitos mensuales (cálculos de intereses compuestos). En el Ejemplo 10.30 vimos un programa en C que genera el valor futuro de una suma de dinero dada durante un período específico de tiempo para varias tasas de interés. El programa fue estructurado originalmente de manera que ilustrará cómo una función es pasada como argumento a otra función. En particular, `main` pasa otra función, bien `md1`, `md2` o `md3` a `tabla`, la cual genera una tabla de valores futuros para las tasas de interés.

Presentamos ahora dos variaciones de este programa. La primera versión utiliza llamadas a funciones directamente desde `main`, mientras que la segunda hace uso de las sustituciones de macros. Aquí está la primera versión.

```
/* cálculos financieros personales, utilizando llamadas a funciones */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>

/* prototipos de funciones */
double md1(double i, int m, double n);
double md2(double i, int m, double n);
double md3(double i, double n);

main() /* calcular el valor futuro de una serie de depósitos mensuales */
{

   enum {A = 1, S = 2, Q = 4, M = 12, D = 360, C} m;
                  /* número de períodos de composición por año */

   int cont;      /* contador del bucle */
   double n;      /* número de años */
   double a;      /* cantidad de cada pago mensual */
   double i;      /* tasa de interés anual */
   double f;      /* valor futuro */
   char frec;     /* indicador de frecuencia de composición */

   /* entrada de datos */
   printf("\nVALOR FUTURO DE UNA SERIE DE DEPOSITOS MENSUALES\n\n");
   printf("Cantidad de cada pago mensual: ");
   scanf("%lf", &a);
   printf("Número de años: ");
   scanf("%lf", &n);

   /* introducir frecuencia de composición */
   do {
            printf("Frecuencia de composición (A, S, Q, M, D, C): ");
            scanf("%1s", &frec);
```

```
            frec = toupper(frec);   /* convertir a mayúsculas */
            if (frec == 'A') {
               m = A;
               printf("\nComposición anual\n");
            }
            else if (frec == 'S') {
               m = S;
               printf("\nComposición semestral\n");
            }
            else if (frec == 'Q') {
               m = Q;
               printf("\nComposición trimestral\n");
            }
            else if (frec == 'M') {
               m = M;
               printf("\nComposición mensual\n");
            }
            else if (frec == 'D') {
               m = D;
               printf("\nComposición diaria\n");
            }
            else if (frec == 'C') {
               m = C;
               printf("\nComposición continua\n");
            }
            else
               printf("\nERROR - Por favor repita\n\n");
   } while (frec != 'A' && frec != 'S' && frec != 'Q' &&
            frec != 'M' && frec != 'D' && frec != 'C');

   /* realizar los cálculos */
   printf("\nTasa de interés     Cantidad futura\n\n");

   for (cont = 1; cont <= 20; ++cont) {
       i = 0.01 * cont;
       if (m == C)
          f = a * md3(i, n);        /* composición continua */
       else if (m == D)
          f = a * md2(i, m, n);     /* composición diaria */
       else
          f = a * md1(i, m, n);     /* composición anual, semestral,
                                       trimestral o mensual */
       printf("     %2d              %.2f\n", cont, f);
   }
}

double md1(double i, int m, double n)
/* depósitos mensuales, composición periódica */
```

```
{
   double factor, razon;

   factor = 1 + i/m;
   razon = 12 * (pow(factor, m*n) - 1) / i;
   return(razon);
}

double md2(double i, int m, double n)
/* depósitos mensuales, composición diaria */

{
   double factor, razon;

   factor = 1 + i/m;
   razon = (pow(factor, m*n) - 1) / (pow(factor, m/12) - 1);
   return(razon);
}

double md3(double i, double n)
/* depósitos mensuales, composición continua */

{
   double razon;

   razon = (exp(i*n) - 1) / (exp(i/12) - 1);
   return(razon);
}
```

Observe que la función `tabla`, que estaba incluida en el programa original, ahora está combinada con `main`, evitando así la necesidad de pasar una función a otra. El programa actual utiliza una enumeración para simplificar un poco la organización interna.

Aquí está la segunda versión, que utiliza sustituciones de macros en lugar de funciones.

```
/* cálculos financieros personales, utilizando sustituciones de ma-
   cros */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>

#define md1(i, m, n) { /* depósitos mensuales, composición periódica */ \
            factor = 1 + i/m;                                           \
            razon = 12 * (pow(factor, m*n) - 1) / i;                    \
        }

#define md2(i, m, n) { /* depósitos mensuales, composición diaria */ \
            factor = 1 + i/m;                                         \
            razon = (pow(factor, m*n) - 1) / (pow(factor, m/12) - 1); \
        }
```

```
#define md3(i, n) { /* depósitos mensuales, composición continua */ \
            razon = (exp(i*n) - 1) / (exp(i/12) - 1);               \
        }

main() /*  calcular el valor futuro de una serie de depósitos mensuales */
{
    enum {A = 1, S = 2, Q = 4, M = 12, D =360, C} m;
                              /* número de períodos de composición por
                                 año */
    int cont;                 /* contador del bucle */
    double n;                 /* número de años */
    double a;                 /* cantidad de cada pago mensual */
    double i;                 /* tasa de interés anual */
    double f;                 /* valor futuro */
    double factor, razon;     /* parámetros internos */
    char frec;                /* indicador de frecuencia de
                                 composición */

    /* entrada de datos */
    printf("\nVALOR FUTURO DE UNA SERIE DE DEPOSITOS MENSUALES\n\n");
    printf("Cantidad de cada pago mensual: ");
    scanf("%lf", &a);
    printf("Número de años: ");
    scanf("%lf", &n);

    /* introducir frecuencia de composición */
    do {
            printf("Frecuencia de composición (A, S, Q, M, D, C): ");
            scanf("%1s", &frec);
            frec = toupper(frec); /* convertir a mayúsculas */
            if (frec == 'A') {
               m = A;
               printf("\nComposición anual\n");
            }
            else if (frec == 'S') {
               m = S;
               printf("\nComposición semestral\n");
            }
            else if (frec == 'Q') {
               m = Q;
               printf("\nComposición trimestral\n");
            }
            else if (frec == 'M') {
               m = M;
               printf("\nComposición mensual\n");
            }
            else if (frec == 'D') {
               m = D;
               printf("\nComposición diaria\n");
            }
```

```
            else if (frec == 'C') {
               m = C;
               printf("\nComposición continua\n");
            }
            else
               printf("\nERROR - Por favor repita\n\n");
   } while (frec != 'A' && frec != 'S' && frec != 'Q' &&
            frec != 'M' && frec != 'D' && frec != 'C');

   /* realizar los cálculos */
   printf("\nTasa de interés    Cantidad futura\n\n");
   for (cont = 1; cont <= 20; ++cont) {
       i = 0.01 * cont;

       if (m == C)
          md3(i, n)      /* composición continua */

       else if (m == D)
          md2(i, m, n)  /* composición diaria */

       else
          md1(i, m, n)  /* composición anual, semestral,
                                          trimestral o mensual */

       f = a * razon;
       printf("    %2d           %.2f\n", cont, f);
   }
}
```

Examine cuidadosamente estos dos programas, comparando el uso de las sustituciones de las macros en lugar de las funciones. En particular, observe la manera en que se accede a las funciones en el primer programa y compárela con las referencias a las macros en el segundo programa (vea la instrucción `if-else` al final de `main`).

Cuando se ejecutan, ambos programas se comportan exactamente de la misma manera que el programa original dado en el Ejemplo 10.30.

Muchos compiladores comerciales de C ofrecen ciertas funciones de biblioteca, en forma de macros o como verdaderas funciones. Las macros están definidas en los archivos de cabecera. El programador puede entonces elegir qué forma es más apropiada para cada aplicación en particular. Pero debe quedar claro que hay ciertas desventajas asociadas con el uso de macros en lugar de funciones, además del potencial incremento significativo de la longitud de un programa. En particular:

1. Cuando se pasan argumentos a una macro, el número de argumentos será comprobado, pero no sus tipos de datos. Así, hay una menor comprobación de errores que con una llamada a una función.
2. Un identificador de macro no está asociado con una dirección; por tanto no puede ser utilizado como un puntero. Así, una macro no puede ser pasada a una función como un

argumento en el mismo sentido que una función puede ser pasada a otra función como un argumento (ver sección 10.9). Además, una macro no puede llamarse a sí misma recursivamente.

3. Hay posibles efectos laterales indeseables con el uso de macros, particularmente cuando hay argumentos en la llamada.

EJEMPLO 14.14. Consideremos la definición de macro

```
#define  raiz(a, b)  sqrt(a*a + b*b)
```

Supongamos ahora que esta macro es utilizada dentro de un programa de la siguiente manera:

```
raiz(a+1, b+2)
```

La intención es, por supuesto, evaluar la fórmula

```
sqrt((a+1)*(a+1) + (b+2)*(b+2))
```

Pero cada aparición de `a` es sustituida por `a+1` (sin paréntesis) y cada aparición de `b` es sustituida por `b+2`. Por tanto, el resultado de la sustitución de la macro será

```
sqrt(a+1*a+1 + b+2*b+2)
```

Esta expresión es equivalente a

```
sqrt(2*a+1 + 3*b+2) = sqrt(2*a + 3*b + 3)
```

que es claramente incorrecta. Sin embargo, la fuente del error puede ser corregida colocando paréntesis adicionales dentro de la definición de macro, esto es,

```
#define  raiz(a, b)  sqrt((a)*(a) + (b)*(b))
```

Un error más sutil ocurre si escribimos

```
raiz(a++, b++)
```

La sustitución de la macro da como resultado la siguiente expresión:

```
sqrt(a*(a+1) + b*(b+1))
```

en vez de

```
sqrt(a*a + b*b)
```

como se pretendía. Éste es un ejemplo de efectos laterales indeseados. Colocando paréntesis adicionales dentro de la definición de la macro no corregiremos estos problemas.

14.5. EL PREPROCESADOR DE C

El preprocesador de C es una colección de instrucciones especiales, llamadas *directivas*, que son ejecutadas al principio del proceso de compilación. Las instrucciones `#include` y `#define` vistas anteriormente en este libro son directivas del preprocesador. Directivas adicionales son `#if`, `#elif`, `#else`, `#endif`, `#ifdef`, `#ifndef`, `#line` y `#undef`. El preprocesador también incluye tres operadores especiales: `defined`, `#` y `##`.

Las directivas del preprocesador generalmente aparecen al principio de un programa, pero esto no es obligatorio. Así, una directiva del preprocesador puede aparecer en cualquier parte dentro de un programa. Pero la directiva tendrá aplicación sólo en la parte del programa que sigue a su aparición.

Para los programadores con poca experiencia, algunas de las directivas del preprocesador son relativamente poco importantes. Por esto, no describiremos cada característica del preprocesador en detalle. A continuación se discuten las características más importantes.

Las directivas `#if`, `#elif`, `#else` y `#endif` son las más usadas. Permiten compilaciones condicionales del programa fuente, dependiendo del valor de una o más condiciones verdadero/falso. A veces se utilizan junto al operador `defined`, que es usado para determinar si una constante simbólica o una macro ha sido definida o no dentro de un programa.

EJEMPLO 14.15. Las siguientes directivas del preprocesador ilustran la compilación condicional de un programa en C. La compilación condicional depende del estado de la constante simbólica `PRIMERPLANO`.

```
#if defined(PRIMERPLANO)
      #define FONDO 0
#else
      #define PRIMERPLANO 0
      #define FONDO 7
#endif
```

Así, si `PRIMERPLANO` ha sido definida, la constante simbólica `FONDO` representará el valor `0`. En otro caso, `PRIMERPLANO` y `FONDO` representarán los valores `0` y `7`, respectivamente.

Esta es otra forma de hacer lo mismo.

```
#ifdef PRIMERPLANO
      #define FONDO 0
#else
      #define PRIMERPLANO 0
      #define FONDO 7
#endif
```

La directiva `#ifdef` es equivalente a `#if defined()`. Análogamente, la directiva `#ifndef` es equivalente a `#if !defined()`, esto es, «si no está definido». Es preferible la primera forma, en la cual el operador `defined` aparece explícitamente.

En cada uno de estos ejemplos, la última directiva es `#endif`. El preprocesador impone que cualquier conjunto de instrucciones que comience con `#if`, `#ifdef` o `#ifndef` debe terminar con `#endif`.

La directiva `#elif` es análoga a una cláusula `else - if` utilizando las instrucciones de control ordinarias de C. Una directiva `#if` puede ir seguida por cualquier número de directivas

#elif, pero sólo puede haber una directiva #else. La aparición de la directiva #else es opcional, determinada por la lógica del programa requerida.

EJEMPLO 14.16. Aquí tenemos otra ilustración de la compilación condicional. En esta situación la compilación condicional dependerá del valor representado por la constante simbólica FONDO.

```
#if FONDO == 7
    #define PRIMERPLANO 0
#elif FONDO == 6
    #define PRIMERPLANO 1
#else
    #define PRIMERPLANO 6
#endif
```

En este ejemplo vemos que PRIMERPLANO representará 0 si FONDO representa 7, y PRIMERPLANO representará 1 si FONDO representa 6. De otra forma, PRIMERPLANO representará 6.

La directiva #undef anula la definición de una constante simbólica o un identificador de macro; es decir, niega el efecto de la directiva #define que puede haber aparecido anteriormente en el programa.

EJEMPLO 14.17. El siguiente ejemplo ilustra el uso de la directiva #undefine dentro de un programa en C.

```
#define PRIMERPLANO 7
#define FONDO 0

main()
{
   . . . . .
   #undef PRIMERPLANO
   . . . . .
   #undef FONDO
   . . . . .
}
```

Las constantes simbólicas PRIMERPLANO y FONDO son definidas por las dos primeras directivas. Estas definiciones son luego negadas por las directivas #undef, cuando aparecen posteriormente en el programa. Antes de las directivas #undef, cualquier referencia a PRIMERPLANO y FONDO será asociada con los valores 7 y 0, respectivamente. Después de las directivas #undef, cualquier referencia a PRIMERPLANO y FONDO será ignorada.

El operador «de cadena» # permite convertir un argumento formal dentro de una definición de una macro en una cadena de caracteres. Si un argumento formal en la definición de una macro es precedida por este operador, el correspondiente argumento real será automáticamente encerrado entre comillas. Los caracteres de espaciado consecutivos dentro del argumento real serán reemplazados por un solo espacio en blanco, y cualquier carácter especial, tal como ', " y \, será reemplazado por su correspondiente secuencia de escape; esto es, \', \" y \\. Además, la cadena resultante será automáticamente concatenada (combinada) con cualquier cadena adyacente.

EJEMPLO 14.18. Aquí tenemos una ilustración del uso del operador «de cadena», #.

```
#define muestra(texto) printf(#texto "\n")

main()
{
  . . . . .
  muestra(Por favor no dormir en clase.);
  . . . . .
  muestra(Por favor - ¡no ronque durante la 'explicación'!);
}
```

Dentro de main, las macros equivalen a

```
printf("Por favor no dormir en clase.\n");
```

y

```
printf("Por favor - ¡no ronque durante la \'explicación\'!\n");
```

Observe que cada argumento real es convertido a cadena de caracteres dentro de la función printf. Cada argumento es concatenado con un carácter de nueva línea (\n), que es escrita como una cadena separada dentro de la definición de la macro. Observe también que los espacios en blanco consecutivos en el segundo argumento son reemplazados por un solo blanco y cada apóstrofo (') es sustituido por su correspondiente secuencia de escape (\ ').

La ejecución del programa produce la siguiente salida:

```
Por favor no dormir en clase.

Por favor - ¡no ronque durante la 'explicación'!
```

El operador «de concatenación» ## hace que los elementos individuales dentro de una macro sean concatenados, formando un solo elemento. Las reglas que rigen el uso de este operador son un poco complicadas. Sin embargo, vamos a ilustrar el propósito general del operador de concatenación con el siguiente ejemplo.

EJEMPLO 14.19. Un programa en C contiene la siguiente definición de macro:

```
#define  muestra(i)  printf("x" #i " = %f\n", x##i)
```

Supongamos que la macro es accedida escribiendo

```
muestra(3);
```

El resultado será

```
printf("x3 = &f\n", x3);
```

Así, la expresión x##i se convierte en x3, ya que 3 es el valor real del argumento i.

Observe que este ejemplo ilustra el uso de ambos operadores, el operador de cadena (#) y el operador de concatenación (##).

Para más información sobre el uso del preprocesador de C, consulte el manual de referencia del programador de su compilador particular.

CUESTIONES DE REPASO

14.1. ¿Qué es una enumeración? ¿Cómo se define una enumeración?
14.2. ¿Qué son las constantes de enumeración? ¿De qué forma están escritas?
14.3. Mencionar las reglas para asignar nombres a las constantes de enumeración.
14.4. Mencionar las reglas para asignar valores numéricos a las constantes de enumeración. ¿Qué valores por omisión son asignados a las constantes de enumeración?
14.5. ¿Pueden dos o más constantes de enumeración tener el mismo valor numérico? Explicarlo.
14.6. ¿Qué son las variables de enumeración? ¿Cómo son declaradas?
14.7. ¿De qué formas pueden ser procesadas las variables de enumeración? ¿Qué restricciones se aplican al procesamiento de las variables de enumeración?
14.8. ¿Qué ventaja existe al usar variables de enumeración en un programa?
14.9. Mencionar las reglas para asignar valores iniciales a las variables de enumeración. Comparar las respuestas con las de la Cuestión 14.4.
14.10. La mayoría de los programas en C reconocen dos argumentos formales en la definición de la función `main`. ¿Cómo son llamados tradicionalmente? ¿Cuáles son sus respectivos tipos de datos?
14.11. Describir la información presentada por cada uno de los argumentos formales en la función `main`. ¿Es información pasada explícitamente a cada uno de los argumentos?
14.12. Cuando se pasan parámetros al programa desde la línea de órdenes, ¿cómo es iniciada la ejecución del programa? ¿Dónde aparecen los parámetros?
14.13. ¿Para qué pueden servir los parámetros de la línea de órdenes cuando se ejecuta un programa que involucra el uso de archivos de datos?
14.14. Las funciones de biblioteca discutidas en anteriores capítulos de este libro son miembros de algunas categorías de funciones de biblioteca. Describir cada categoría en términos generales.
14.15. Describir, en términos generales, otras categorías adicionales de funciones de biblioteca que vienen con la mayoría de los compiladores comerciales de C. ¿Cuál es el propósito de cada categoría?
14.16. ¿Qué es una macro? Mencionar las similitudes y diferencias entre macros y funciones.
14.17. ¿Cómo se define una macro multilínea?
14.18. Describir el uso de argumentos dentro de una macro.
14.19. ¿Cuál es la principal ventaja de usar macros en vez de funciones? ¿Cuál es la principal desventaja? ¿Qué otras desventajas hay?
14.20. Mencionar las otras directivas del preprocesador distintas de `#include` y `#define`. Indicar el propósito de las más usadas.

14.21. ¿Cuál es el ámbito de una directiva del preprocesador dentro de un archivo de programa?

14.22. Mencionar los operadores especiales del preprocesador `#` y `##`. ¿Cuál es el propósito de cada uno de ellos?

14.23. ¿Qué se entiende por compilación condicional? En términos generales, ¿cómo se realiza la compilación condicional? ¿Qué directivas del preprocesador son válidas para este propósito?

PROBLEMAS

14.24. Definir un tipo de enumeración llamado `indicadores` que tenga los siguientes miembros: `primero`, `segundo`, `tercero`, `cuarto` y `quinto`.

14.25. Definir una variable de enumeración llamada `suceso` del tipo `indicadores` (ver problema anterior).

14.26. Definir dos variables de enumeración, llamadas `soprano` y `bajo`, cuyos nombres sean los siguientes: `do`, `re`, `mi`, `fa`, `sol`, `la` y `si`. Asignar los siguientes valores enteros a estos miembros:

```
do      1
re      2
mi      3
fa      4
sol     5
la      6
si      7
```

14.27. Definir un tipo de enumeración llamado `dinero` que tenga los siguientes miembros: `penique`, `niquel`, `dime`, `cuarto`, `medio` y `dolar`. Asignar los siguientes valores enteros a estos miembros:

```
penique    1
niquel    15
dime      10
cuarto    25
medio     50
dolar    100
```

14.28. Definir una variable de enumeración llamada `moneda` del tipo `dinero` (ver problema anterior). Asignar el valor inicial `dime` a `moneda`.

14.29. En la siguiente declaración de enumeración, determinar el valor de cada miembro:

```
enum  brujula  {norte = 2, sur, este = 1, oeste};
```

14.30. Determinar el valor asociado con cada una de las siguientes variables de enumeración (ver problema anterior):

```
enum  brujula  mover_1 = sur, mover_2 = norte;
```

14.31. Explicar el propósito del siguiente esquema de programa.

```
int tantos = 0;
enum brujula mover:

. . . . .

switch(mover) {

case norte:
        tantos += 10;
        break;

case sur:
        tantos += 20;
        break;

case este:
        tantos += 30;
        break;

case oeste:
        tantos += 40;
        break;

default:
        printf("ERROR - Por favor, pruebe de nuevo\n");
}
```

14.32. A continuación se muestra el esquema de un programa en C.

```
main(int argc, char *argv[])
{
   . . . . .
}
```

a) Supongamos que el programa objeto compilado es almacenado en un archivo llamado `demo.exe` y se escribe la siguiente orden para iniciar la ejecución del programa:

```
demo depurar rapido
```

Determinar el valor de `argc` y los elementos no vacíos de `argv`.

b) Supongamos que la línea de órdenes está escrita como

```
demo "depurar rapido"
```

¿Cómo afecta este cambio a los valores de `argc` y `argv`?

14.33. Describir el propósito del programa en C mostrado a continuación.

```
#include <stdio.h>
#include <string.h>
```

```
main(int argc, char *argv[])
{
   char letra[80];
   int cont, marca;

   for (cont = 0; (letra[con] = getchar()) != '\n'; ++cont)
       ;
   marca = cont;

   for (cont = 0; cont < marca; ++cont)
       if (strcmp(argv[1], "mayuscula") == 0)
           putchar(toupper(letra[cont]));
       else if (strcmp(argv[1], "minuscula") == 0)
               putchar(tolower(letra[cont]));
       else {
          puts("ERROR EN LA LINEA DE ORDENES - PRUEBE DE NUEVO");
          break;
   }
}
```

14.34. Considerar el programa mostrado a continuación, que lee una línea de texto de un archivo de datos existente, la muestra en pantalla y la escribe en un nuevo archivo de datos.

```
/* leer una línea de texto de un archivo de datos existente,
   mostrarla en pantalla y escribirla en un nuevo archivo */

#include <stdio.h>

#define NULL 0

main(int argc, char *argv[])
{
   FILE *fpt1, *fpt2;

   char c;

   /* abrir el archivo de datos antiguo solo para lectura */
   if ((fpt1 = fopen(argv[1], "r")) == NULL)
      printf("\nERROR - No se puede abrir el archivo indicado\n");

   else {
      /* leer, mostrar y escribir cada carácter del archivo de
         datos antiguo */
      fpt2 = fopen(argv[2], "w");
      do {
            putchar(c = getc(fpt1));
            putc(c, fpt2);
      } while (c != '\n');

   /* cerrar archivos de datos */
   fclose(fpt1);
   fclose(fpt2);
}
```

Supongamos que el programa se almacena en un archivo llamado `transfer.exe`, el archivo de datos antiguo se llama `datos.ant` y el nuevo archivo `datos.nue`. ¿Cómo debe ser escrita la línea de órdenes para ejecutar este programa?

14.35. Escribir una constante simbólica o una definición de macro para cada una de las siguientes situaciones. No incluir argumentos a menos que el problema lo indique.

a) Definir una constante simbólica `PI` para representar el valor `3.1415927`.
b) Definir una macro llamada `AREA` que calcule el área de un círculo en función de su radio. Usar en el cálculo la constante `PI`, definida en la parte *a*).
c) Reescribir la macro del problema anterior de modo que el radio se exprese como un argumento.
d) Definir una macro llamada `CIRCUNFERENCIA` que calcule la circunferencia de un círculo en función de su radio. Usar en el cálculo la constante `PI`, definida en la parte *a*).
e) Reescribir la macro del problema anterior de modo que el radio se exprese como un argumento.
f) Escribir una macro multilínea llamada `interes` que evalúe la fórmula del interés compuesto

$$F = P(1 + i)^n$$

donde F es la cantidad futura de dinero que se acumulará al cabo de n años, P es el principal (la cantidad original de dinero), $i = 0.01r$ y r es la tasa de interés anual expresada en porcentaje.
Evaluar i en una línea de la macro, y evaluar F en otra línea separada. Suponer que todos los símbolos representan cantidades en coma flotante.
g) Reescribir la macro definida en el problema anterior de modo que P, r y n sean expresadas como argumentos.
h) Escribir una macro llamada `max` que utilice el operador condicional (`? :`) para determinar el máximo de `a` y `b`, donde `a` y `b` son cantidades enteras.
i) Reescribir la macro del problema anterior de modo que `a` y `b` se expresen como argumentos.

14.36. Explicar el propósito de cada uno de los siguientes grupos de directivas del preprocesador.

a)
```
#if  !defined(INDICADOR)
     #define  INDICADOR  1
#endif
```

b)
```
#if  defined(PASCAL)
     #define  BEGIN {
     #define  END   }
#endif
```

c)
```
#ifdef  CELSIUS
     #define  temperatura(t)  0.5555555 * (t - 32)
#else
     #define  temperatura(t)  1.8 * t + 32
#endif
```

d)
```
#ifndef  DEPURAR
     #define  salida  printf("x = %f\n", x)
#elif  NIVEL == 1
     #define  salida  printf("i = %d   y = %f\n", i, y[i])
#else
     #define  salida  for (cont = 1; cont <= n; ++cont)   \
                          printf("i = %d   y = %f\n", i, y[i])
#endif
```

e)
```
#if  defined(DEPURAR)
     #undef  DEPURAR
#endif
```

f)
```
#ifdef  COMPROBACION_ERROR
     #define  mensaje(linea)  printf(#linea)
#endif
```

g)
```
#if  defined(COMPROBACION_ERROR)
     #define  mensaje(n)  printf("%s\n",  mensaje##n)
#endif
```

14.37. Escribir una o más directivas del preprocesador para cada una de las situaciones siguientes:

a) Si la constante simbólica `LOGICA` ha sido definida, definir las constantes simbólicas `VERDADERO` y `FALSO` tales que sus valores sean `1` y `0`, respectivamente, y negar las dos definiciones de las constantes simbólicas `SI` y `NO`.
b) Si `indicador` tiene valor `0`, definir la constante simbólica `COLOR` con valor `1`. En otro caso, si el valor de `indicador` es menor que `3`, definir `COLOR` con valor `2`; y si el valor de indicador es mayor o igual que `3`, definir `COLOR` con valor `3`.
c) Si la constante simbólica `TAMANO` tiene el mismo valor que la constante simbólica `AMPLIO`, definir la constante simbólica `ANCHO` con el valor `132`; en otro caso, definir `ANCHO` con valor de `80`.
d) Usar el operador de «cadena» para definir una macro llamada `error(texto)` que escribirá `texto` como una cadena.
e) Usar el operador de «concatenación» para definir una macro llamada `error(i)` que escribirá el valor de la variable de cadena `errori` (por ejemplo, `error1`).

14.38. Familiarizarse con la biblioteca de funciones y con los archivos de cabecera que acompañan a su compilador particular de C. ¿Hay algunas funciones que están disponibles tanto como macros y como verdaderas funciones?

14.39. ¿Incluye la biblioteca de funciones de su compilador de C rutinas gráficas o de control de procesos? ¿Están incluidas otras rutinas especiales? En ese caso, ¿cuáles son?

PROBLEMAS DE PROGRAMACIÓN

14.40. Modificar el programa dado en el Ejemplo 14.13 (valor futuro de depósitos mensuales) tal que acepte un parámetro de la línea de órdenes que indique la frecuencia de composición. El parámetro de la línea de órdenes debe ser un carácter seleccionado de `A`, `S`, `Q`, `M`, `D` o `C` (mayúscula o minúscula), como se explicó en el ejemplo.

14.41. Modificar el programa dado en el Ejemplo 6.22 (solución de una ecuación algebraica) de modo que `indicador` sea una variable de enumeración cuyo valor sea `verdadero` o `falso`.

14.42. Modificar el programa dado en el Ejemplo 6.32 (búsqueda de palíndromos) de modo que `indicador` sea una variable de enumeración cuyo valor sea `verdadero` o `falso`.

14.43. Modificar el programa dado en el Ejemplo 7.9 (el mayor de tres enteros) de modo que la función `maximo` sea escrita como una macro multilínea.

14.44. Modificar el programa dado en el Ejemplo 7.10 (cálculo de factoriales) de modo que la función `factorial` sea escrita como una macro multilínea.

14.45. Modificar el programa dado en el Ejemplo 7.11 (juego de dados «Craps») de modo que la función `lanzar` sea escrita como una macro multilínea. ¿Puede usarse de forma efectiva una variable de enumeración en este problema?

14.46. Escribir un programa completo en C para resolver el problema descrito en el Problema 7.42 (raíces de una ecuación de segundo grado). Incluir una variable de enumeración dentro del programa.

14.47. Escribir un programa completo en C para resolver el problema descrito en el Problema 9.46 (nombres de países y sus capitales). Usar una variable de enumeración para distinguir entre dos opciones del programa (encontrar el nombre de la capital para un país especificado, o encontrar el país cuya capital ha sido especificada).

14.48. Modificar el programa dado en el Ejemplo 10.28 (presentación del día del año) de modo que use una variable de enumeración para los meses del año.

14.49. Escribir un programa completo en C para resolver el problema descrito en el Problema 11.67 (mantenimiento de estadísticas de equipos de fútbol/béisbol). Incluir una variable de enumeración para distinguir entre fútbol y béisbol.

14.50. Escribir un programa completo en C para resolver el problema descrito en el Problema 11.71 (una calculadora RPN). Incluir una variable de enumeración para identificar los tipos de operaciones aritméticas que pueden ser realizadas por la computadora.

14.51. Repetir el Problema 14.50, utilizando macros en lugar de funciones.

14.52. Modificar el programa dado en cada uno de los siguientes ejemplos de modo que los nombres de archivos requeridos sean introducidos como parámetros de la línea de órdenes:

a) Ejemplo 12.3 (creación de un archivo de datos).
b) Ejemplo 12.4 (lectura de un archivo de datos).

14.53. Modificar el programa dado en cada uno de los siguientes ejemplos de modo que los nombres de archivos requeridos sean introducidos como parámetros de la línea de órdenes. Utilizar una variable de enumeración para representar las condiciones internas cierto/falso dentro de cada programa.

a) Ejemplo 12.5 (creación de un archivo de datos que contiene registros de clientes).
b) Ejemplo 12.6 (actualización de un archivo de datos que contiene registros de clientes).
c) Ejemplo 12.7 (creación de un archivo de datos sin formato que contiene registros de clientes).
d) Ejemplo 12.8 (actualización de un archivo de datos sin formato que contiene registros de clientes).

14.54. Escribir un programa completo en C para resolver cada uno de los siguientes problemas.

a) Problema 12.50 (editor de texto orientado a líneas).
b) Problema 12.51 (mantenimiento de estadísticas de equipos de fútbol/béisbol en un archivo de datos).

Para cada programa introducir los nombres de archivos requeridos como parámetros de la línea de órdenes.

14.55. Cada uno de los siguientes problemas requiere que uno o más valores numéricos sean especificados como parámetros de la línea de órdenes. Usar las funciones de biblioteca `atoi` y `atof` para convertir los parámetros de la línea de órdenes en enteros y valores en coma flotante, respectivamente.

a) Escribir un programa completo en C para resolver el problema descrito en el Problema 7.49(*a*) (generación recursiva de polinomios de Legendre). Introducir los valores de `n` y `x` como parámetros de la línea de órdenes.

b) Escribir un programa completo en C para resolver el problema descrito en el Problema 7.49(*b*) (calcular recursivamente la suma de `n` números en coma flotante). Introducir el valor de `n` como un parámetro de la línea de órdenes (pero la introducción de datos se hace en modo interactivo como antes).

c) Escribir un programa completo en C para resolver el problema descrito en el Problema 7.49(*c*) (calcular recursivamente los primeros `n` términos de una serie). Introducir el valor de `n` como un parámetro de la línea de órdenes.

d) Escribir un programa completo en C para resolver el problema descrito en el Problema 7.49(*d*) (calcular de forma recursiva el producto de `n` números en coma flotante). Introducir el valor de `n` como un parámetro de la línea de órdenes. (Sin embargo, la introducción de los números en coma flotante se hace en modo interactivo como antes.)

e) Modificar el programa dado en el Ejemplo 8.4 (búsqueda del máximo) de la siguiente manera:

i) Introducir valores para `CNST`, `a` y `b` como parámetros de la línea de órdenes.
ii) Escribir la función `curva` como una macro.

f) Modificar el programa dado en el Ejemplo 8.7 (generación de números de Fibonacci) de modo que el valor de `n` sea introducido como un parámetro de la línea de órdenes.

g) Modificar el programa dado en el Ejemplo 9.13 (reordenación de una lista de números) de modo que el valor de `n` sea introducido como un parámetro de la línea de órdenes.

h) Modificar el programa dado en el Ejemplo 9.19 (suma de dos tablas de números) de modo que los valores de `nfilas` y `ncolumnas` sean introducidos como parámetros de la línea de órdenes.

14.56. Escribir un programa completo en C para generar la tabla descrita en el Problema 9.43. Usar una macro para evaluar la fórmula

$$y = 2e^{-0.1t} \text{ sen } 0.5t$$

14.57. Escribir un programa completo en C para generar la tabla descrita en el Problema 9.44. Usar una macro para evaluar la fórmula

$$F/P = (1 + i/100)^n$$

14.58. Escribir un programa completo en C para resolver el problema descrito en el Problema 7.44 (evaluar la fórmula $y = x^n$). Usar una macro multilínea en lugar de la función para realizar la potenciación.

APÉNDICE A

Sistemas de representación de números

Decimal	*Binario*	*Octal*	*Hexadecimal*
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Observe que hay ocho dígitos octales y 16 hexadecimales. Los dígitos octales van de 0 a 7; los dígitos hexadecimales van de 0 a F.

Cada dígito octal es equivalente a tres dígitos binarios (3 bits), y cada dígito hexadecimal es equivalente a cuatro dígitos binarios (4 bits). Por tanto, los números octales y hexadecimales proporcionan una forma conveniente y concisa para representar patrones binarios. Por ejemplo, el patrón binario 10110111 se representa en hexadecimal como B7. Para ver más claramente esta relación se reordenan los bits en grupos de cuatro y cada grupo se representa mediante un dígito hexadecimal; por ejemplo, 1011 0111 se representa con B 7.

Análogamente, este mismo patrón binario (10110111) puede representarse en octal como 267. Para ver más claramente esta relación se le añade un cero a la izquierda (de modo que el número de bits del patrón sea múltiplo de 3), se reordenan los bits en grupos de tres y se representa cada grupo como un dígito octal; por ejemplo, 010 110 111 se representa con 2 6 7.

La mayoría de las computadoras utilizan números hexadecimales para representar patrones de bits, si bien algunas usan números octales para este propósito.

APÉNDICE B

Secuencias de escape

Carácter	*Secuencia de escape*	*Valor ASCII*
sonido (alerta)	`\a`	007
retroceso (espacio atrás)	`\b`	008
tabulador horizontal	`\t`	009
nueva línea (cambio de línea)	`\n`	010
tabulador vertical	`\v`	011
nueva página	`\f`	012
retorno de carro	`\r`	013
comillas (")	`\"`	034
apóstrofo (')	`\'`	039
interrogación (?)	`\?`	063
barra invertida (\)	`\\`	092
nulo	`\0`	000
número octal	`\ooo` (o representa un dígito octal)	

Normalmente no se permiten más de tres dígitos octales.

Ejemplos: `\5, \005, \123, \177`

número hexadecimal `\xhh` (h representa un dígito hexadecimal)

Normalmente se permite cualquier número de dígitos hexadecimales.

Ejemplos: `\x5, \x05, \x53, \x7f`

La mayoría de los compiladores permiten que el apóstrofo (') y la interrogación (?) aparezcan dentro de una cadena de caracteres constante bien como un carácter ordinario o bien como una secuencia de escape.

APÉNDICE C

Resumen de operadores

<table>
<tr><th>Grupo de precedencia</th><th>Operadores</th><th>Asociatividad</th></tr>
<tr><td>función, array, miembro de estructura, puntero a miembro de estructura</td><td><code>() [] . -></code></td><td>I → D</td></tr>
<tr><td>operadores unarios</td><td><code>- ++ -- ! ~</code>
<code>* & sizeof (tipo)</code></td><td>D → I</td></tr>
<tr><td>multiplicación, división y resto aritméticos</td><td><code>* / %</code></td><td>I → D</td></tr>
<tr><td>suma y resta aritméticas</td><td><code>+ -</code></td><td>I → D</td></tr>
<tr><td>operadores de desplazamiento a nivel de bits</td><td><code><< >></code></td><td>I → D</td></tr>
<tr><td>operadores relacionales</td><td><code>< <= > >=</code></td><td>I → D</td></tr>
<tr><td>operadores de igualdad</td><td><code>== !=</code></td><td>I → D</td></tr>
<tr><td>y a nivel de bits</td><td><code>&</code></td><td>I → D</td></tr>
<tr><td>o exclusiva a nivel de bits</td><td><code>^</code></td><td>I → D</td></tr>
<tr><td>o a nivel de bits</td><td><code>|</code></td><td>I → D</td></tr>
<tr><td>y lógica</td><td><code>&&</code></td><td>I → D</td></tr>
<tr><td>o lógica</td><td><code>||</code></td><td>I → D</td></tr>
<tr><td>operador condicional</td><td><code>? :</code></td><td>D → I</td></tr>
<tr><td>operadores de asignación</td><td><code>= += -= *= /= %=</code>
<code>&= ^= |= <<= >>=</code></td><td>D → I</td></tr>
<tr><td>operador coma</td><td><code>,</code></td><td>I → D</td></tr>
</table>

Nota: Los grupos de precedencia están ordenados de mayor a menor precedencia. Algunos compiladores de C también incluyen un operador más unario (+) para complementar el operador menos unario (-). Por tanto, una expresión con el más unario es equivalente al valor de su operando: por ejemplo, +v tiene el mismo valor que v.

APÉNDICE D

Tipos de datos y reglas de conversión de datos

Tipo de dato	*Descripción*	*Requisitos típicos de memoria*
`int`	Cantidad entera	2 bytes o 1 palabra (varía de una computadora a otra)
`short`	Cantidad entera corta (puede contener menos dígitos que `int`)	2 bytes o 1 palabra (varía de una computadora a otra)
`long`	Cantidad entera larga (puede contener más dígitos que `int`)	1 o 2 palabras (varía de una computadora a otra)
`unsigned`	Cantidad entera sin signo (no negativa) (la cantidad máxima permisible es aproximadamente el doble que `int`)	2 bytes o 1 palabra (varía de una computadora a otra)
`char`	Carácter	1 byte
`signed char`	Carácter, con valores en el rango de `-128` a `+127`	1 byte
`unsigned char`	Carácter, con valores en el rango de `0` a `255`	1 byte
`float`	Número en coma flotante (un número que contiene un punto decimal y/o un exponente)	1 palabra
`double`	Número en coma flotante en doble precisión (mas cifras significativas y un exponente que puede ser mayor en valor absoluto)	2 palabras
`long double`	Número en coma flotante en doble precisión (puede tener mayor precisión que un `double`)	2 o más palabras (varía de una computadora a otra)
`void`	Tipo de dato especial para funciones que no devuelven ningún valor	(no aplicable)
`enum`	Constante de enumeración (tipo especial de `int`)	2 bytes o 1 palabra (varía de una computadora a otra)

Nota: El cualificador `unsigned` puede aparecer con `short int` o con `long int`, por ejemplo, `unsigned short int` (o `unsigned short`), o `unsigned long int` (o `unsigned long`).

REGLAS DE CONVERSIÓN

Estas reglas se aplican a operaciones aritméticas entre dos operadores con distintos tipos de datos. Puede existir alguna variación de una versión de C a otra.

1. Si uno de los operandos es `long double`, el otro será convertido a `long double` y el resultado será un `long double`.
2. En otro caso, si uno de los operandos es `double`, el otro será convertido a `double` y el resultado será `double`.
3. En otro caso, si uno de los operandos es `float`, el otro será convertido a `float` y el resultado será `float`.
4. En otro caso, si uno de los operandos es `unsigned long int`, el otro será convertido a `unsigned long int` y el resultado será `unsigned long int`.
5. En otro caso, si uno de los operandos es `long int` y el otro es `unsigned int`, entonces:
   a) Si `unsigned int` se puede convertir a `long int`, el operando `unsigned int` será convertido y el resultado será `long int`.
   b) En otro caso, ambos operandos serán convertidos a `unsigned long int` y el resultado será `unsigned long int`.
6. En otro caso, si uno de los operandos es `long int`, el otro será convertido a `long int` y el resultado será `long int`.
7. En otro caso, si uno de los operandos es `unsigned int`, el otro será convertido a `unsigned int` y el resultado será `unsigned int`.
8. Si no se puede aplicar ninguna de las condiciones anteriores, entonces ambos operandos serán convertidos a `int` (si es necesario) y el resultado será `int`.

Tenga en cuenta que algunas versiones de C convierten automáticamente todos los operandos en coma flotante a doble precisión.

REGLAS DE ASIGNACIÓN

Si los dos operandos en una expresión de asignación son de tipos distintos, entonces el valor del operando de la derecha será automáticamente convertido al tipo del operando de la izquierda. La expresión de asignación completa será de este mismo tipo. Además:

1. Un valor en coma flotante puede truncarse si se asigna a un identificador entero.
2. Un valor en doble precisión puede ser redondeado si se asigna a un identificador en coma flotante (simple precisión).
3. Una cantidad entera puede ser alterada si se asigna a un identificador de entero más corto o a un identificador de carácter (algunos de los bits más significativos pueden perderse).

APÉNDICE E

El conjunto de caracteres ASCII

Valor ASCII	*Carácter*	*Valor ASCII*	*Carácter*	*Valor ASCII*	*Carácter*	*Valor ASCII*	*Carácter*
0	NUL	32	espacio en blanco	64	@	96	`
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	'	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	\	124	|
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	DEL

Los primeros 32 caracteres y el último son caracteres de control. Normalmente no se visualizan. Sin embargo, algunas versiones de C (en algunas computadoras) soportan caracteres gráficos especiales para estos valores ASCII. Por ejemplo, 001 puede representar el carácter ☺, 002 puede representar ☻, y así sucesivamente.

APÉNDICE F

Resumen de instrucciones de control

Instrucción	*Forma general*	*Ejemplo*
break	break;	for (n = 1; n <= 100; ++n) { scanf("%f", &x); if (x < 0) { printf("ERROR - NEGATIVO"); break; } . . . . . }
continue	continue;	for (n = 1; n <= 100; ++n) { scanf("%f", &x); if (x < 0) { printf("ERROR - NEGATIVO"); continue; } . . . . . }
do-while	do *instrucción* while (*expresión*);	do printf("%d\n" digito++); while (digito <= 9);
for	for (*exp 1*; *exp 2*; *exp 3*) *instrucción*	for (digito = 0; digito <= 9; ++digito printf("%d\n", digito);
goto	goto *etiqueta*; . . . . . *etiqueta*: *instrucción*	if (x < 0) goto indicador; . . . . . indicador: printf("ERROR");
if	if (*expresión*) *instrucción*	if (x < 0) printf("%f", x);

Instrucción	*Forma general*	*Ejemplo*
if - else	if (*expresión*) *instrucción 1* else *instrucción 2*	if (estado == 'S') tasa = 0.20 * pago; else tasa = 0.14 * pago;
return	return *expresión*;	return (n1 + n2);
switch	switch (*expresión*) { case *expresión 1*: *instrucción 1* *instrucción 2* . . . . . *instrucción m* break; case *expresión 2*: *instrucción 1* *instrucción 2* . . . . . *instrucción n* break; . . . . . default: *instrucción 1* *instrucción 2* . . . . . *instrucción k* }	switch (eleccion = getchar()) { case 'R': printf("ROJO"); break; case 'B': printf("BLANCO"); break; case 'A': printf("AZUL"); break; default: printf("ERROR"); }
while	while (*expresión*) *instrucción*	while (digito <= 9) printf("%d\n", digito++);

APÉNDICE G

Caracteres de conversión más usados de `scanf` y `printf`

Caracteres de conversión de `scanf`

Carácter de conversión	*Significado*
`c`	el dato es un carácter
`d`	el dato es un entero decimal
`e`	el dato es un valor en coma flotante
`f`	el dato es un valor en coma flotante
`g`	el dato es un valor en coma flotante
`h`	el dato es un entero corto
`i`	el dato es un entero decimal, hexadecimal u octal
`o`	el dato es un entero octal
`s`	el dato es una cadena de caracteres seguido por un espacio en blanco (el carácter nulo `\0` se añade automáticamente al final)
`u`	el dato es un entero decimal sin signo
`x`	el dato es un entero hexadecimal
`[. . .]`	el dato es una cadena de caracteres que puede contener espacios en blanco

Un *prefijo* puede preceder a ciertos caracteres de conversión.

Prefijo	*Significado*
`h`	dato corto (entero corto o entero sin signo corto)
`l`	dato largo (entero largo, entero largo sin signo o real en doble precisión)
`L`	dato largo (real en doble precisión largo)

Ejemplo:

```
int a;
short b;
long c;
unsigned d;
double x;
char cad[80];
```

```
scanf("%5d  %3hd  %12ld  %12lu  %15lf", &a, &b, &c, &d, &x);

scanf("%[^\n]", cad);
```

Caracteres de conversión de `printf`

Carácter de conversión	*Significado*
`c`	el dato se muestra como un carácter
`d`	el dato se muestra como un entero decimal
`e`	el dato se muestra como un valor en coma flotante con exponente
`f`	el dato se muestra como un valor en coma flotante sin exponente
`g`	el dato se muestra como un valor en coma flotante usando la conversión de tipo `e` o `f`, dependiendo del valor; no se muestran ceros no significativos ni el punto decimal si no es significativo
`i`	el dato se muestra como un entero decimal con signo
`o`	el dato se muestra como un entero octal, sin el cero inicial
`s`	el dato se muestra como una cadena de caracteres
`u`	el dato se muestra como un entero decimal sin signo
`x`	el dato se muestra como un entero hexadecimal, sin el `0x` del principio

Observe que algunos de estos caracteres se interpretan de modo diferente que con la función `scanf`.

Un *prefijo* puede preceder a ciertos caracteres de conversión.

Prefijo	*Significado*
`h`	dato corto (entero corto o entero corto sin signo)
`l`	dato largo (entero largo, entero largo sin signo o real en doble precisión)
`L`	dato largo (real en doble precisión largo)

Ejemplo:

```
int a;
short b;
long c;
unsigned d;
double x;
char cad[80];

printf("%5d  %3hd  %12ld  %12lu  %15.7le\n", a, b, c, d, x);

printf("%40s\n", cad);
```

Indicadores

Indicador	*Significado*
-	El dato es justificado a la izquierda dentro del campo (los espacios en blanco necesarios para rellenar la longitud de campo mínima se añadirán *detrás* del dato en vez de *delante*).
+	Un signo (+ o –) precederá a cada valor numérico con signo. Sin este indicador, sólo los valores negativos estarán precedidos por un signo.
0	Produce la aparición de ceros iniciales en vez de espacios en blanco. Se aplica sólo a datos que están justificados a la derecha en campos cuya longitud mínima es mayor que el dato.
	(*Nota*: algunos compiladores consideran el indicador cero como una parte de la especificación de la longitud de campo en vez de como un indicador real. Esto asegura que el 0 se procese el último si están presentes múltiples indicadores.)
' ' (espacio en blanco)	Un espacio en blanco precederá a cada valor positivo. Este indicador es anulado por el indicador + si ambos están presentes.
# (con conversiones de tipo o- y x-)	Hace que los datos octales y hexadecimales sean precedidos por 0 y 0x, respectivamente.
# (con conversiones de tipo e-, f- y g-)	Hace que esté presente el punto decimal en todos los números en coma flotante, incluso si el número no tiene decimales. También trunca los ceros no significativos en una conversión del tipo g-.

Ejemplo:

```
int a;
short b;
long c;
unsigned d;
double x;

printf("%+5d  %+5hd  %+12ld  %-12lu  %#15.7le\n", a, b, c, d, x);
```

APÉNDICE H

Funciones de biblioteca más usadas

Función	*Tipo*	*Propósito*	*Archivo* include
abs(i)	int	Devuelve el valor absoluto de i.	stdlib.h
acos(d)	double	Devuelve el arco coseno de d.	math.h
asin(d)	double	Devuelve el arco seno de d.	math.h
atan(d)	double	Devuelve la arco tangente de d.	math.h
atan(d1, d2)	double	Devuelve el arco tangente de d1/d2.	math.h
atof(s)	double	Convierte la cadena s a una cantidad en doble precisión.	stdlib.h
atoi(s)	int	Convierte la cadena s a un entero.	stdlib.h
atol(s)	long	Convierte la cadena s a un entero largo.	stdlib.h
calloc(u1, u2)	void*	Reserva memoria para un array de u1 elementos, cada uno de u2 bytes . Devuelve un puntero al principio del espacio reservado.	malloc.h o stdlib.h
ceil(d)	double	Devuelve un valor redondeado por exceso al siguiente entero mayor.	math.h
cos(d)	double	Devuelve el coseno de d.	math.h
cosh(d)	double	Devuelve el coseno hiperbólico de d.	math.h
difftime(11, 12)	double	Devuelve la diferencia de tiempo 11-12, donde 11 y 12 representan el tiempo transcurrido después de un tiempo base (ver la función time).	time.h
exit(u)	void	Cierra todos los archivos y búffers y termina el programa. El valor de u es asignado por la función para indicar el estado de terminación.)	stdlib.h
exp(d)	double	Eleva e a la potencia d (e=2.7182818... es la base del sistema de logaritmos naturales (Neperianos)).	math.h

Función	*Tipo*	*Propósito*	*Archivo* include
fabs(d)	double	Devuelve el valor absoluto de d.	math.h
fclose(f)	int	Cierra el archivo f. Devuelve 0 si el archivo se ha cerrado con éxito.	stdio.h
feof(f)	int	Determina si se ha encontrado un fin de archivo. Si es así, devuelve un valor distinto de cero; en otro caso devuelve 0	stdio.h
fgetc(f)	int	Lee un carácter del archivo f.	stdio.h
fegts(s, i, f)	char*	Lee una cadena s, con i caracteres, del archivo f	stdio.h
floor(d)	double	Devuelve un valor redondeado por defecto al entero menor más cercano.	math.h
fmod(d1, d2)	double	Devuelve el resto de d1/d2 (con el mismo signo que d1).	math.h
fopen(s1, s2)	file*	Abre un archivo llamado s1, del tipo s2. Devuelve un puntero al archivo.	stdio.h
fprintf(f,...)	int	Escribe datos en el archivo f (el resto de los argumentos son complicados; ver Apéndice G).	stdio.h
fputc(c, f)	int	Escribe un carácter en el archivo f.	stdio.h
fputs(s, f)	int	Escribe una cadena de caracteres en el archivo f.	stdio.h
fread(s,i1,i2,f)	int	Lee i2 elementos, cada uno de tamaño i1 bytes, desde el archivo f hasta la cadena s.	stdio.h
free(p)	void	Libera un bloque de memoria reservada cuyo principio está indicado por p.	malloc.h o stdlib.h
fscanf(f,...)	int	Lee datos del archivo f (el resto de los argumentos son complicados; ver Apéndice G).	math.h
fseek(f, l, i)	int	Mueve el puntero al archivo f una distancia de l bytes desde la posición i (i puede representar el principio del archivo, la posición actual del puntero o el fin del archivo).	stdio.h
ftell(f)	long int	Devuelve la posición actual del puntero dentro del archivo f.	stdio.h
fwrite(s,i1,i2,f)	int	Escribe i2 elementos, cada uno de tamaño i1 bytes, desde la cadena s hasta el archivo f.	stdio.h

Función	*Tipo*	*Propósito*	*Archivo* include
getc(f)	int	Lee un carácter del archivo f.	stdio.h
getchar()	int	Lee un carácter desde el dispositivo de entrada estándar.	stdio.h
gets(s)	char*	Lee una cadena de caracteres desde el dispositivo de entrada estándar.	stdio.h
isalnum(c)	int	Determina si el argumento es alfanumérico. Devuelve un valor distinto de cero si es cierto; en otro caso devuelve 0.	ctype.h
isalpha(c)	int	Determina si el argumento es alfabético. Devuelve un valor distinto de cero si es cierto; en otro caso devuelve 0.	ctype.h
isascii(c)	int	Determina si el argumento es un carácter ASCII. Devuelve un valor distinto de cero si es cierto; en otro caso devuelve 0.	ctype.h
iscntrl(c)	int	Determina si el argumento es un carácter ASCII de control. Devuelve un valor distinto de cero si es cierto; en otro caso devuelve 0.	ctype.h
isdigit(c)	int	Determina si el argumento es un dígito decimal. Devuelve un valor distinto de cero si es cierto; en otro caso devuelve 0.	ctype.h
isgraph(c)	int	Determina si el argumento es un carácter ASCII gráfico (hex 0x21-0x7e; octal 041-176). Devuelve un valor distinto de cero si es cierto; en otro caso devuelve 0.	ctype.h
islower(c)	int	Determina si el argumento es una minúscula. Devuelve un valor distinto de cero si es cierto; en otro caso devuelve 0.	ctype.h
isodigit(c)	int	Determina si el argumento es un dígito octal. Devuelve un valor distinto de cero si es cierto; en otro caso devuelve 0.	ctype.h
isprint(c)	int	Determina si el argumento es un carácter ASCII imprimible (hex 0x20-0x7e; octal 040-176). Devuelve un valor distinto de cero si es cierto; en otro caso devuelve 0.	ctype.h
ispunct(c)	int	Determina si el argumento es un carácter de puntuación. Devuelve un valor distinto de cero si es cierto; en otro caso devuelve 0.	ctype.h

Función	*Tipo*	*Propósito*	*Archivo* include
isspace(c)	int	Determina si el argumento es un espacio en blanco. Devuelve un valor distinto de cero si es cierto; en otro caso devuelve 0.	ctype.h
isupper(c)	int	Determina si el argumento es una mayúscula. Devuelve un valor distinto de cero si es cierto; en otro caso devuelve 0.	ctype.h
isxdigit(c)	int	Determina si el argumento es un dígito hexadecimal. Devuelve un valor distinto de cero si es cierto; en otro caso devuelve 0.	ctype.h
labs(l)	long int	Devuelve el valor absoluto de l.	math.h
log(d)	double	Devuelve el logaritmo natural de d.	math.h
log10(d)	double	Devuelve el logaritmo (en base 10) de d.	math.h
malloc(u)	void*	Reserva u bytes de memoria. Devuelve un puntero al principio del espacio reservado.	stdlib.h
pow(d1, d2)	double	Devuelve d1 elevado a la potencia d2.	math.h
printf(...)	int	Escribe datos en dispositivo de salida estándar (los argumentos son complicados; ver Apéndice G).	stdio.h
putc(c, f)	int	Escribe un carácter en el archivo f.	stdio.h
putchar(c)	int	Escribe un carácter en el dispositivo de salida estándar.	stdio.h
puts(s)	int	Escribe una cadena de caracteres en el dispositivo de salida estándar.	stdio.h
rand()	int	Devuelve un entero positivo aleatorio.	stdlib.h
rewind(f)	void	Mueve el puntero al principio del archivo f.	stdio.h
scanf(...)	int	Lee datos en dispositivo de entrada estándar (los argumentos son complicados; ver Apéndice G).	stdio.h
sin(d)	double	Devuelve el seno de d.	math.h
sinh(d)	double	Devuelve el seno hiperbólico de d.	math.h
sqrt(d)	double	Devuelve la raíz cuadrada de d.	math.h
srand(u)	void	Inicializa el generador de números aleatorios.	stdlib.h
strcmp(s1, s2)	int	Compara dos cadenas de caracteres lexicográficamente. Devuelve un valor negativo si s1 < s2; 0 si s1 y s2 son idénticas; y un valor positivo si s1 > s2.	string.h

Función	*Tipo*	*Propósito*	*Archivo* include
strcmpi(s1, s2)	int	Compara dos cadenas de caracteres lexicográficamente, sin diferenciar mayúsculas de minúsculas. Devuelve un valor negativo si s1 < s2; 0 si s1 y s2 son idénticas; y un valor positivo si s1 > s2.	string.h
strcpy(s1, s2)	char*	Copia la cadena de caracteres s2 en la cadena s1.	string.h
strlen(s)	int	Devuelve el número de caracteres de una cadena.	string.h
strset(c, s)	char*	Pone todos los caracteres de s a c (excluyendo el carácter nulo del final \0).	string.h
system(s)	int	Pasa la orden al sistema operativo. Devuelve 0 si la orden se ejecuta correctamente; en otro caso devuelve un valor distinto de cero, típicamente -1.	string.h
tan(d)	double	Devuelve la tangente de d.	math.h
tanh(d)	double	Devuelve la tangente hiperbólica de d.	math.h
time(p)	long int	Devuelve el número de segundos transcurridos después de un tiempo base designado.	time.h
toascii	int	Convierte el valor del argumento a ASCII.	ctype.h
tolower	int	Convierte una letra a minúscula.	ctype, o stdlib.h
toupper	int	Convierte una letra a mayúscula.	ctype, o stdlib.h

Notas: *tipo* se refiere al tipo de dato de la cantidad devuelta por la función. Un asterisco (*) denota un puntero.

- c denota un argumento de tipo carácter
- d denota un argumento en doble precisión
- f denota un argumento archivo
- i denota un argumento entero
- l denota un argumento entero largo
- p denota un argumento puntero
- s denota un argumento cadena de caracteres
- u denota un argumento entero sin signo

La mayoría de los compiladores comerciales de C vienen acompañados de muchas más funciones de biblioteca. Consulte el manual de referencia de su compilador particular para una información más detallada de las funciones anteriores y para tener una lista de las funciones adicionales.

Respuestas a problemas seleccionados

Capítulo 1

1.31. *a*) Este programa escribe el mensaje `¡Bienvenido a la Informática!`. El programa no contiene ninguna variable. La línea que contiene `printf` es una instrucción de salida. No hay instrucciones de entrada ni de asignación.

b) Este programa también escribe el mensaje `¡Bienvenido a la Informática!`. El programa no contiene ninguna variable. (`MENSAJE` es una constante simbólica, no una variable.) La línea que contiene `printf` es una instrucción de salida. No hay instrucciones de entrada ni de asignación.

c) Este programa calcula el área de un triángulo a partir de su base y su altura. Las variables son `base`, `altura` y `area`. Las instrucciones `printf - scanf` alternativas generan la entrada interactiva. La instrucción `printf` final es una instrucción de salida. La instrucción que comienza con `area =` es una instrucción de asignación.

d) Este programa calcula el salario neto (descontando impuestos), dado el salario bruto y el tanto por ciento de impuestos (que es una constante, `14%`). Las variables son `bruto`, `impuesto` y `neto`. Las instrucciones `printf - scanf` iniciales generan la entrada interactiva. Las dos instrucciones `printf` finales son instrucciones de salida. Las instrucciones que contienen `impuesto =` y `neto =` son instrucciones de asignación.

e) Este programa utiliza una función para calcular el menor de dos números enteros. Las variables son `a`, `b` y `min`. Los pares de instrucciones `printf - scanf` alternativas generan la entrada interactiva. La instrucción `printf` final es una instrucción de salida. La instrucción `min = menor(a, b)` hace referencia a la función llamada `menor`. Esta función contiene una instrucción `if - else` que devuelve la menor de las dos cantidades a la parte principal del programa.

f) Este programa procesa `n` pares de cantidades enteras y determina la cantidad menor de cada par. Se utiliza un bucle `for` para procesar los múltiples pares de cantidades enteras. En cuanto al resto, el programa es análogo al mostrado en la parte (*e*).

g) Este programa procesa un número indeterminado de pares de cantidades enteras y determina la menor de cada par. La ejecución continúa hasta que se introduce un par de ceros. Se utiliza un bucle `while` para procesar los múltiples pares de cantidades enteras. En cuanto al resto, el programa es análogo al mostrado en la parte (*f*).

h) Este programa procesa un número indeterminado de pares de cantidades enteras y determina la menor de cada par. Los valores originales y los mínimos correspondientes se almacenan en los arrays `a`, `b` y `min`. Cada array puede almacenar hasta 100 valores enteros.

Una vez que se han introducido todos los datos y se han determinado todos los mínimos, el número de conjuntos de datos es «apuntado» mediante la instrucción `n = --i`; se utiliza entonces un bucle `for` para visualizar los datos. En cuanto al resto, el programa es análogo al mostrado en la parte (*g*).

Capítulo 2

2.39. *a*) Válido.

b) Un identificador debe comenzar con una letra.

c) Válido.

d) `return` es una palabra reservada.
e) Un identificador debe comenzar con una letra.
f) Válido.
g) No están permitidos los espacios en blanco.
h) Válido.
i) No está permitido el signo menos.
j) Un identificador debe comenzar con una letra o un carácter de subrayado.

2.40. *a*) Distinto *c*) Idéntico *e*) Distinto
b) Distinto *d*) Distinto *f*) Distinto

2.41. *a*) Válido (real).
b) Carácter ilegal (`,`).
c) Válido (real).
d) Válido (real).
e) Válido (entero decimal).
f) Válido (entero largo).
g) Válido (real).
h) Carácter ilegal (espacio en blanco).
i) Válido (constante octal).
j) Caracteres ilegales (`C`, `D`, `F`) en una constante octal. Si se interpreta como constante hexadecimal, es necesario incluir `x` o `X` (por ejemplo, `0X1CDF`).
k) Válido (entero largo hexadecimal).
l) Carácter ilegal (`h`).

2.42. *a*) Válido.
b) Válido.
c) Válido.
d) Las secuencias de escape se deben escribir con una barra invertida.
e) Válido.
f) Válido.
g) Válido.
h) Válido (secuencia de escape del carácter nulo).
i) Una constante de carácter no puede constar de varios caracteres.
j) Válido (secuencia de escape en octal). Notar que el octal `52` es equivalente al decimal `42`. En el conjunto de caracteres ASCII, el valor representa un asterisco (*).

2.43. *a*) Una constante de cadena de caracteres debe encontrarse encerrada entre comillas dobles.
b) Válido.
c) Faltan las comillas finales.
d) Válido.
e) Válido.
f) Válido.
g) Las comillas y (opcionalmente) el apóstrofo dentro de las cadenas de caracteres se deben expresar como secuencias de escape; por ejemplo, `"El profesor dijo, \"Por favor, no se duerman en clase\"";`

2.44. *a*)
```
int p, q;
float x, y, z;
char a, b, c;
```
b)
```
float raiz1, raiz2;
long cont;
short indicador;
```

c)
```
int indice;
unsigned num_cliente;
double bruto, impuesto, neto;
```

d)
```
char actual, ultimo;
unsigned cont;
float error;
```

e)
```
char primero, ultimo;
char mensaje[80];
```

2.45. *a*)
```
float a = -8.2, b = 0.005;
int x = 129, y = 87, z = -22;
char c1 = 'w', c2 = '&';
```

b)
```
double d1 = 2.88e-8, d2 = -8.4e5;
int u = 0711, v = 0xffff;
```

c)
```
long grande = 123456789L;
double c = 0.3333333333;
char eol ='\n';
```

d)
```
char mensaje[] = "ERROR";
```

2.46. *a*) Restar el valor de b al de a.
b) Multiplicar por a la suma de los valores de b y c.
c) Multiplicar por a la suma de los valores de b y c. Asignar el resultado a d.
d) Determinar si el valor de a es mayor o igual que el de b. El resultado será cierto o falso, representado por los valores 1 (cierto) y 0 (falso).
e) Dividir el valor de a por 5 y determinar si el resto es igual a cero. El resultado será cierto o falso.
f) Dividir el valor de b por el valor de c y determinar si el valor de a es menor que el cociente. El resultado será cierto o falso.
g) Decrementar el valor de a; por ejemplo, decrementar el valor de a en 1.

2.47. *a*) Instrucción de expresión
b) Instrucción de control que contiene una instrucción compuesta. (La instrucción compuesta está encerrada entre llaves.)
c) Instrucciónde control
d) Instrucción compuesta que contiene instrucciones de expresión y una instrucción de control.
e) Instrucción compuesta que contiene una instrucción de expresión y una instrucción de control. La instrucción de control contiene dos instrucciones compuestas.

2.48. *a*) `#define FACTOR -18`
b) `#define ERROR 0.0001`
c)
```
#define BEGIN {
#define END }
```
d) `#define NOMBRE "Adrián"`
e) `#define EOLN '\n'`
f) `#define COSTE "$19.95"`

Capítulo 3

3.36. *a*) 6
b) 45
c) 2

d) 2
e) –1
f) 3
g) –4
h) 0 (porque `b / c` es cero)
i) –1
j) –16

3.37. *a*) 7.1
b) 49
c) 2.51429
d) La operación resto no está definida para operandos en coma flotante.
e) –5.17647
f) –2.68571
g) 20.53333
h) 1.67619

3.38. *a*) 69
b) 79
c) 51
d) 3
e) 98
f) 6
g) 100
h) 63
i) 159
j) 2703

3.39. *a*) entero
b) coma flotante (algunas versiones de C lo convertirán a doble precisión)
c) doble precisión
d) entero largo
e) coma flotante (o doble precisión)
f) entero
g) entero largo
h) entero
i) entero largo

3.40. *a*) 14
b) 18
c) –466.6667
d) –13
e) 9
f) 9
g) 4
h) 1.005
i) –1.01
j) 0
k) 0
l) 1

m) 0
n) 1
o) 1
p) 0
q) 1
r) 0.01
s) 1
t) 1
u) 0
v) 0
w) 0
x) 1
y) 1
z) 0

3.41. *a*) k=13
b) z=–0.005
c) i=5
d) k=0
e) k=99
f) z=1.0
g) b=100, a=100 (Observe que 100 es el valor de 'd' en el conjunto de caracteres ASCII)
h) j=1, i=1
i) k=0, z=0.0
j) z=0.005, k=0 [comparar con *i*) anterior]
k) i=10
l) i=-0.015
m) x=0.010
n) i=1
o) i=3
p) i=11
q) k=8
r) k=5
s) k=0.005
t) z=0.0
u) a='c'
v) i=3

3.42. *a*) Devuelve el valor absoluto de la expresión entera (i - 2 * j).
b) Devuelve el valor absoluto de la expresión en coma flotante (x + y).
c) Determina si el carácter representado por c es un carácter ASCII imprimible.
d) Determina si el carácter representado por c es un dígito decimal.
e) Convierte el carácter representado por c a mayúscula.
f) Redondea el valor de x al valor entero superior próximo.
g) Redondea el valor de (x + y) al valor entero inferior próximo.
h) Determina si el carácter representado por c es una minúscula.
i) Determina si el carácter representado por j es una mayúscula.
j) Devuelve el valor de e^x.
k) Devuelve el logaritmo natural de x.

l) Devuelve la raíz cuadrada de la expresión `(x*x + y*y)`.
m) Determina si el valor de la expresión `(10 * j)` se puede interpretar como un carácter alfanumérico.
n) Determina si el valor de la expresión `(10 * j)` se puede interpretar como un carácter alfabético.
o) Determina si el valor de la expresión `(10 * j)` se puede interpretar como un carácter ASCII.
p) Convierte el valor de la expresión `(10 * j)` a un carácter ASCII.
q) Divide el valor de `x` por el de `y`, y devuelve el resto con el mismo signo que `x`.
r) Convierte el carácter ASCII cuyo código es `65` a minúscula.
s) Determina la diferencia entre el valor de `x` y el valor de `y`, y la eleva a la potencia `3.0`.
t) Evalúa la expresión `(x - y)` y devuelve su seno.
u) Devuelve el número de caracteres de la cadena `"hola"`.
v) Devuelve la posición de la primera aparición de la letra `o` en la cadena `"hola"`.

3.43. *a*) 2
b) 0.005
c) 1
d) 0
e) `'D'`
f) 1.0
g) 0.0
h) 0.0
i) –1.0
j) 1
k) 0
l) 1.005013
m) –5.298317
n) 0.005
o) 0.011180
p) 1
q) 0
r) 1
s) `'2'`
t) 0.005
u) `'a'`
v) 3.375e-6
w) 0.014999
x) 5
y) 1 (0 indica la primera posición)
z) 1.002472

Capítulo 4

4.50. *a*)
```
a = getchar();
b = getchar();
c = getchar();
```
b)
```
putchar(a);
putchar(b);
putchar(c);
```

4.51. *a*) `scanf("%c%c%c", &a, &b, &c);`
o `scanf("%c %c %c", &a, &b, &c);`

b) `printf("%c%c%c", a, b, c);`
o `printf("%c %c %c", a, b, c);`

4.52. *a*)
```
for (cont = 0; cont < 60; ++cont)
    texto[cont] = getchar();
```
b)
```
for (cont = 0; cont < 60; ++cont)
    putchar(texto[cont]);
```
(Nota: se supone que `cont` es una variable entera.)

4.53.
```
for (cont = 0; (texto[cont] = getchar()) != '\n'; ++cont)
    ;
```

4.54. `scanf("%[^\n]", texto);`

El método utilizado en el Problema 4.53 indica el número de caracteres que se han leído.

4.55. *a*) `scanf("%d %d %d", &i, &j, &k);`
b) `scanf("%d %o %x", &i, &j, &k);`
c) `scanf("%x %x %o", &i, &j, &k);`

4.56. *a*) `scanf("%6d %6d %6d", &i, &j, &k);`
b) `scanf("%8d %8o %8x", &i, &j, &k);`
c) `scanf("%7x %7x %7o", &i, &j, &k);`

4.57. *a*) A `a` se le asignará un entero largo decimal con una longitud de campo máxima de `12`; a `b` se le asignará un entero corto decimal con una longitud de campo máxima de `5`; a `c` y `d` se les asignarán cantidades de doble precisión con longitud de campo máxima de `15`.
b) A `a` se le asignará un entero largo hexadecimal con una longitud de campo máxima de `10`; a `b` se le asignará un entero corto octal con una longitud de campo máxima de `6`; a `c` se le asignará un entero corto sin signo con una longitud de campo máxima de `6`; a `d` se le asignará un entero largo sin signo con una longitud de campo máxima de `14`.
c) A `a` se le asignará un entero largo decimal con una longitud de campo máxima de `12`; a `b` se le asignará un entero corto decimal con una longitud de campo máxima no especificada; a `c` y `d` se les asignarán cantidades en coma flotante con longitud de campo máxima de `15`.
d) A `a` se le asignará un entero decimal con una longitud de campo máxima de `8`; a continuación se leerá otro entero decimal y no se asignará a ninguna variable; a `c` y `d` se les asignarán cantidades de doble precisión con longitud de campo máxima de `12`.

4.58. *a*) `scanf("%d %d %e %le", &i, &j, &x, &dx);`
o `scanf("%d %d %f %lf", &i, &j, &x, &dx);`
b) `scanf("%d %ld %d %f %u", &i, &ix, &j, &x, &u);`
c) `scanf("%d %u %c", &i, &u, &c);`
d) `scanf("%c %f %lf %hd", &c, &x, &dx, &s);`
o `scanf("%c %e %le %hd", &c, &x, &dx, &s);`

4.59. *a*) `scanf("%4d %4d %8e %15le", &i, &j, &x, &dx);`
o `scanf("%4d %4d %8f %15lf", &i, &j, &x, &dx);`

b) `scanf("%5d %12ld %5d %10f %5u", &i, &ix, &j, &x, &u);`
c) `scanf("%6d %6u %c", &i, &u, &c);`
d) `scanf("%c %9f %16lf %6hd", &c, &x, &dx, &s);`
o `scanf("%c %9e %16le %6hd", &c, &x, &dx, &s);`

4.60. `scanf("%s", texto);`

4.61. `scanf("%[ abcdefghijklmnopqrstuvwxyz\n"]", texto);`

4.62. `scanf("%[ ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890$\n"]", texto);`

4.63. `scanf("%[^*]", texto);`

4.64. *a*) `$*@` (los caracteres sin separar por espacios)
b) `$ * @` (uno o más espacios en blanco entre los caracteres)
c) `$   *    @` (uno o más caracteres de espaciado entre los caracteres de entrada)
d) `$ * @` (uno o más espacios en blanco entre los caracteres. Pueden aparecer otros caracteres de espaciado entre ellos)
e) `$*@` (los caracteres sin separar por espacios)

4.65. *a*) `12  -8  0.011  -2.2e6`
b) `12  -8  0.011  -2.2e6`
c) `12  -8  0.011  -2.2e6`
d) `12  -8  0.011  -2.2e6`

Observe que no se puede exceder la longitud de campo máxima especificada y que uno o más espacios en blanco deben separar las cantidades numéricas sucesivas. La representación de valores en coma flotante más conveniente es la que se muestra, independientemente de los caracteres de conversión particulares de cada función `scanf`.

4.66. *a*) `printf("%d %d %d", i, j, k);`
b) `printf("%d %d", (i + j), (i - k));`
c) `printf("%f %d", sqrt(i + j), abs(i - k));`

4.67 *a*) `printf("%3d %3d %3d", i, j, k);`
b) `printf("%5d %5d", (i + j), (i - k));`
c) `printf("%9f %7d", sqrt(i + j), abs(i - k));`

4.68 *a*) `printf("%f %f %f", x, y, z);`
b) `printf("%f %f", (x + y), (x - z));`
c) `printf("%f %f", sqrt(x + y), fabs(x - z);`

Nota: se podría utilizar también la conversión tipo `e` o tipo `g`, por ejemplo,

`printf("%e %e %e", x, y, z);`

4.69. *a*) `printf("%6f %6f %6f", x, y, z);`
b) `printf("%8f %8f", (x + y), (x - z));`
c) `printf("%12f %9f", sqrt(x + y), fabs(x - z);`

4.70. *a*) `printf("%6e %6e %6e", x, y, z);`
b) `printf("%8e %8e", (x + y), (x - z));`
c) `printf("%12e %9e", sqrt(x + y), fabs(x - z);`

En cada caso, los valores numéricos incluirán exponentes.

4.71.
a) `printf("%8.4f %8.4f %8.4f", x, y, z);`
b) `printf("%9.3f %9.3f", (x + y), (x - z));`
c) `printf("%12.4f %10.4f", sqrt(x + y), fabs(x - z);`

4.72.
a) `printf("%12.4e %12.4e %12.4e", x, y, z);`
b) `printf("%14.5e %14.5e", (x + y), (x - z));`
c) `printf("%12.7e %15.7e", sqrt(x + y), fabs(x - z);`

4.73.
a) `printf("%o %o %x %x", a, b, c, d);`
b) `printf("%o %x", (a + b), (c - d));`

4.74.
a) `printf("%d %d %g %g", i, j, x, dx);`
b) `printf("%d %ld %d %g %u", i, ix, j, x, u);`
c) `printf("%d %u %c", i, u, c);`
d) `printf("%c %g %g %ld", c, x, dx, ix);`

Nota: se puede utilizar conversión tipo `e` en lugar de la tipo `g`.

4.75.
a) `printf("%4d %4d %14.8e %14.8e", i, j, x, dx);`
b) `printf("%4d\n %4d\n %14.8e\n %14.8e", i, j, x, dx);`
c) `printf("%5d %12ld %5d %10.5f %5u", i, ix, j, x, u);`
d) `printf("%5d %12ld %5d\n\n %10.5f %5u", i, ix, j, x, u);`
e) `printf("%6d   %6u   %c", i, u, c);`
f) `printf("%5d %5u %11.4f", j, u, x);`
g) `printf("%-5d %-5u %-11.4f", j, u, x);`
h) `printf("%+5d %5u %+11.4f", j, u, x);`
i) `printf("%05d %05u %11.4f", j, u, x);`
j) `printf("%5d %5u %#11.4f", j, u, x);`

4.76.
a) `printf("%8o %8d %8x", i, j, k);`
b) `printf("%-8o %-8d %-8x", i, j, k);`
c) `printf("%#8o %08d %#8x", i, j, k);`

4.77.
a)
```
12345 -13579 -24680 123456789 -2222 5555
```
b)
```
12345 -13579 -24680
123456789 -2222 5555
```
c)
```
   12345   -13579   -24680
      123456789    -2222     5555
```
d)
```
12345     -13579
-24680    123456789
-2222     5555
```
e)
```
+12345    -13579
-24680       +123456789
-2222      5555
```
f)
```
00012345 -0013579
-0024680 000000123456789
-0002222 00005555
```

4.78.

a) `12345 abcd9 77777`
b) `12345 abcd9 77777`
c) `   12345    abcd9     77777`
d) `12345    abcd9     77777`
e) `  +12345    abcd9     77777`
f) `00012345  0xabcd9   077777`

4.79.

a) `2.500000 0.000500 3000.000000`
b) `2.500000 0.000500 3000.000000`
c) `2.500000 0.000500 3000.000000`
d) `  2.5000    0.0005 3000.0000`
e) `   2.500     0.001 3000.000`
f) `2.500000e+000 5.000000e-004 3.000000e+003`
g) `2.500000e+000 5.000000e-004 3.000000e+003`
h) `2.500000e+000 5.000000e-004 3.000000e+003`
i) `2.5000e+000  5.0000e-004  3.0000e+003`
j) `2.50e+000 5.00e-004 3.00e+003`
k) `2.500000 0.000500 3000.000000`
l) `+2.500000 +0.000500 +3000.000000`
m) `2.500000 0.000500 3000.000000`
n) `2.500000 0.000500 3000.000000`
o) `2.5 0.0005 3000`
p) `2.500000 0.000500 3000.000000`

4.80.

a) `A B C`
b) `ABC`
c) `  A    B    C`
d) `  A   B   C`
e) `c1=A c2=B c3=C`

4.81.

a) `printf("%s", texto);`
b) `printf("%.8s", texto);`
c) `printf("%13.8s", texto);`
d) `printf("%-13.8s", texto);`

4.82.

a) `Programar en C puede ser una actividad enormemente creativa.`
b) `Programar en C puede ser una actividad enormemente creativa.`
c) `Programar en C pue`
d) `           Program`
e) `Program`

4.83.

a)
```
printf("Por favor, introduce tu nombre: ");
scanf("%[^\n]", nombre);
```
b)
```
printf("x1 = %4.1f    x2 = %4.1f", x1, x2);
```
c)
```
printf("Por favor, introduce el valor de a: ");
scanf("%d", &a);
printf("Por favor, introduce el valor de b: ");
scanf("%d", &b);
printf("\nLa suma es %d", (a + b));
```

La última instrucción también se podría escribir así:

```
printf("\n%s %d", "La suma es", (a + b));
```

Capítulo 5

5.30. *a*)

```
/* programa "¡HOLA!" */

#include <stdio.h>

main()
{
  printf("%s", "¡HOLA!");
}
```

b)

```
/* programa "BIENVENIDO - SEAMOS AMIGOS" */

#include <stdio.h>

main()
{
  char nombre[20];

  printf("%s", "HOLA, ¿COMO TE LLAMAS?");
  scanf("%[^\n]", nombre);
  printf("\n\n%s%s\n%s", "BIENVENIDO ", nombre, "SEAMOS AMIGOS");
}
```

c)

```
/* conversión de temperaturas - fahrenheit a celsius */

#include <stdio.h>

main()
{
  float c, f;

  printf("%s", "Introduce la temperatura en grados F: ");
  scanf("%f", &f);
  c = (5. / 9.) * (f - 32.)
  printf("\n%s%5.1f", "El valor correspondiente de C es: ", c);
}
```

d)

```
/* problema de la hucha */

#include <stdio.h>

main()
{
  int medios, cuartos, dimes, niqueles, peniques;
  float dolares;

  printf("%s", "¿Cuantos medios dólares? ");
  scanf("%d", &medios);
  printf("%s", "¿Cuantos cuartos? ");
  scanf("%d", &cuartos);
  printf("%s", "¿Cuantos dimes? ");
```

```
    scanf("%d", &dimes);
    printf("%s", "¿Cuantos níqueles? ");
    scanf("%d", &niqueles);
    printf("%s", "¿Cuantos peniques? ");
    scanf("%d", &peniques);
    dolares = 0.5 * medios +0.25 * cuartos + 0.1 * dimes +
                              0.05 * niqueles + 0.01 * peniques;

    printf("\n%s%6.2f%s", "El total es ", dolares, " dólares");
  }
```

e)
```
/* volumen y área de una esfera */

#include <stdio.h>

#define PI 3.1415927

main()
{
  float radio, volumen, area;

  printf("%s", "Por favor introduce el valor del radio: ");
  scanf("%f", &radio);

  volumen = (4. / 3.) * PI * radio * radio * radio;
  area = 4. * PI * radio * radio;

  printf("\n%s%.3e\n%s%.3e", "El volumen es ", volumen,
                             "El área es ", area);
}
```

f)
```
/* masa de aire de un neumático */

#include <stdio.h>

main()
{
  float p, v, m, t;

  printf("%s", "Introduce el volumen en pies cúbicos: ");
  scanf("%f", &v);
  printf("%s", "Introduce el valor de la presión en psi: ");
  scanf("%f", &p);
  printf("%s", "Introduce la temperatura en grados F: ");
  scanf("%f", &t);

  m = (p * v) / (0.37 * (t + 460.));
  printf("\nMasa de aire: %g libras", m);
}
```

g) /* codificación de una palabra de 5 letras */

```
#include <stdio.h>

main()
{
   char c1, c2, c3, c4, c5;

   printf("%s", "Por favor, introduce una palabra de 5 letras: ");
   scanf("%c%c%c%c%c", &c1, &c2, &c3, &c4, &c5);
   printf("%c%c%c%c%c", (c1-30), (c2-30), (c3-30), (c4-30), (c5-30));
}
```

h) /* decodificación de una palabra de 5 letras */

```
#include <stdio.h>

main()
{
   char c1, c2, c3, c4, c5;

   printf("%s", "Introduce la palabra de 5 letras codificada: ");
   scanf("%c%c%c%c%c", &c1, &c2, &c3, &c4, &c5);
   printf("%c%c%c%c%c", (c1+30), (c2+30), (c3+30), (c4+30),
                        (c5+30));
}
```

i) /* codificar y decodificar una línea de texto */

```
#include <stdio.h>

main()
{
   int cont, aux;
   char texto[80];

   /* leer y codificar la línea de texto */
   printf("%s", "Por favor, introduce una línea de texto: \n");
   for (cont = 0; (texto[cont] = getchar() - 30) != ('\n' -
30);        ++cont)
      ;
   aux = cont;

   /* escribir el texto codificado */
   printf("\nTexto codificado:\n");
   for (cont = 0; cont < aux; ++cont)
      putchar(texto[cont]);

   /* decodificar y escribir, recuperando el texto original */
   printf("\n\nTexto decodificado (original):\n");
   for (cont = 0; cont < aux; ++cont)
      putchar(texto[cont] + 30);
}
```

j)
```
/* intercambiar las mayúsculas y minúsculas de una línea de
   texto */

#include <stdio.h>
#include <ctype.h>

main()
{
    int cont, aux;
    char c, texto[80];

    /* leer la línea de entrada */

    printf("%s", "Por favor, introduce una línea de texto:\n");
    for (cont = 0; (texto[cont] = getchar()) != '\n'; ++cont)
        ;
    aux = cont;

    /* escribir la línea de salida */

    for (cont = 0; cont < aux; ++cont)  {
        c = islower(texto[cont]) ? toupper(texto[cont])
                                 :tolower(texto[cont]);
        putchar(c);
    }
}
```

Capítulo 6

6.43. Si el valor absoluto de x es menor que el de xmin, entonces el valor de xmin es asignado a x si x tiene valor positivo, y si el valor de x es negativo o igual a cero, se le asigna a x el valor de -xmin. Ésta no es una instrucción compuesta y no hay instrucciones compuestas incluidas.

6.44.
1) El fragmento del programa es una instrucción compuesta.
2) La instrucción do-while, que está incluida en el fragmento del programa, contiene una instrucción compuesta.
3) La instrucción if, que está incluida en la instrucción do-while, contiene una instrucción compuesta.

6.45. *a*)
```
suma = 0;
i = 2;
while (i < 100)  {
    suma += i;
    i += 3;
}
```

b)
```
suma = 0;
i = 2;
do  {
      suma += i;
      i += 3;
} while (i < 100);
```

c)
```
suma = 0;
for (i = 2; i < 100; i +=3)
    suma += i;
```

6.46. *a*)
```
suma = 0;
i = ncom;
while (i <= nfin)  {
    suma += i;
    i += n;
}
```

b)
```
suma = 0;
i = ncom;
do  {
      suma += i;
      i += n;
} while (i <= nfin);
```

c)
```
suma = 0;
for (i = ncom; i <= nfin; i +=n)
    suma += i;
```

6.47. *a*)
```
cont = 0;
while (cont < n)  {
    printf("%d ", texto[cont]);
    ++cont;
}
```

o

```
cont = 0;
while (cont < n)
    printf("%d ", texto[cont++]);
```

b)
```
cont = 0;
do  {
      printf("%d ", texto[cont]);
      ++cont;
} while (cont < n);
```

o

```
cont = 0;
do
    printf("%d ", texto[cont++]);
while (cont < n);
```

c)
```
for (cont = 0; cont < n; ++cont)
    printf("%d ", texto[cont]);
```

6.48. *a*)
```
cont = 0;
while (texto[cont] != '*')  {
    printf("%d ", texto[cont]);
    ++cont;
}
```

o

```
cont = 0;
while (texto[cont] != '*')
    printf("%d ", texto[cont++]);
```

b)
```
cont = 0;
do  {
       printf("%d ", texto[cont]);
       ++cont;
} while (texto[cont] != '*');
```

o

```
cont = 0;
do
    printf("%d ", texto[cont++]);
while (texto[cont] != '*');
```

c)
```
for (cont = 0; texto[cont] != '*'; ++cont)
    printf("%d ", texto[cont]);
```

6.49. *a*)
```
for (j = 2; j <= 13; ++j)  {
    suma = 0;
    i = 2;
    while (i < 100)  {
        suma += i;
        i += j;
    }
    printf("%d", suma);
}
```

b)
```
for (j = 2; j <= 13; ++j)  {
    suma = 0;
    i = 2;
    do  {
           suma += i;
           i += j;
    } while (i < 100)
    printf("%d", suma);
}
```

c)
```
for (j = 2; j <= 13; ++j)  {
    suma = 0;
    for(i = 2; i < 100; i +=j)
       suma += i;
    printf("%d", suma);
}
```

6.50. *a*)
```
suma = 0;
for (i = 2; i < 100; i += 3)
     suma = (i % 5 == 0) ? += i: += 0;
```

b)
```
suma = 0;
for (i = 2; i < 100; i += 3)
     if (i % 5 == 0) suma += i;
```

6.51. *a*)
```
suma = 0;
for (i = ncom; i <= nfin; i += n)
     suma = (i % k == 0) ? += i: += 0;
```

b)
```
suma = 0;
for (i = ncom; i <= nfin; i +=n)
     if (i % k == 0) suma += i;
```

6.52.
```
letras = digitos = blancos = otros = 0;
for (cont = 0; cont < 80; ++cont)  {
     if ((texto[cont] >= 'a' && texto[cont] <= 'z') ||
         (texto[cont] >= 'A' && texto[cont] <= 'Z'))
            ++letras;
     else if (texto[cont] >= '0' && texto[cont] <= '9')
            ++digitos;
          else if (texto[cont] == ' '  || texto[cont] == '\n' ||
                   texto[cont] == '\t')
                   ++blancos;
               else  ++otros;
}
```

6.53.
```
vocales = consonantes = 0;
for (cont = 0; cont < 80; ++cont)  {
     if (isalpha(texto[cont])
        if (texto[cont] == 'a' || texto[cont] == 'A'  ||
            texto[cont] == 'e' || texto[cont] == 'E'  ||
            texto[cont] == 'i' || texto[cont] == 'I'  ||
            texto[cont] == 'o' || texto[cont] == 'O'  ||
            texto[cont] == 'u' || texto[cont] == 'U')
               ++vocales;
     else ++consonantes;
}
```

El bucle también se puede escribir así:

```
vocales = consonantes = 0;
for (cont = 0; cont < 80; ++cont)  {
     if (isalpha(texto[cont])
        if (tolower(texto[cont]) == 'a' ||
            tolower(texto[cont]) == 'e' ||
            tolower(texto[cont]) == 'i' ||
```

```
            tolower(texto[cont]) == 'o' ||
            tolower(texto[cont]) == 'u')
               ++vocales;
      else ++consonantes;
}
```

6.54.
```
switch (indicador)  {

case 1:  printf("CALOR");
         break;

case 2:  printf("TEMPLADO");
         break;

case 3:  printf("FRIO");
         break;

default: printf("FUERA DE RANGO");
}
```

6.55.
```
switch (color)  {

case 'r':
case 'R':
         printf("ROJO");
         break;

case 'v':
case 'V':
         printf("VERDE");
         break;

case 'a':
case 'A':
         printf("AZUL");
         break;

default:
         printf("NEGRO");
         break;

}
```

6.56.
```
if (temp < 0.)
   printf("HIELO");
else if (temp <= 100.)
        printf("AGUA");
     else
        printf("VAPOR");
```

No se puede utilizar una instrucción `switch` porque:

a) Las comprobaciones involucran cantidades en coma flotante y no enteras.
b) Las comprobaciones involucran intervalos de valores y no valores exactos.

6.57.
```
for (i = 0, j = 79; i < 80; ++i, --j)
    inverso[j] = texto[i];
```

6.58.

a) `0 5 15 30`
`x = 30`

b) `1 2 3 4`
`x = 4`

c) `1 2 3 4`
`x = 4`

d) `1 0 3 2 7 6 13 12 21`
`x = 21`

e) `1 0 3 2 7 6 13 12 21`
`x = 21`

f) `1`
`x = 1`

g) `0 1 3 5 8 12 15 19 24 30`
`x = 30`

h) `0 1 3 6`
`x = 6`

i) `0`
`x = 0`

j) `0 0 2 4 5 9 10 14 14 20`
`x = 20`

k) `1 3 5 7 9 12 14 17 20 23`
`x = 23`

l) `1 6 11 16 21 24 29 32 35 38`
`x = 38`

Capítulo 7

7.32. *a*) `f` acepta un argumento entero y devuelve una cantidad entera.
b) `f` acepta dos argumentos y devuelve una cantidad en doble precisión. El primer argumento es una cantidad en doble precisión y el segundo un entero.
c) `f` acepta tres argumentos y no devuelve nada. El primer argumento es un entero largo, el segundo un entero corto y el tercero un entero sin signo.
d) `f` no acepta ningún argumento y devuelve un carácter.
e) `f` acepta dos argumentos enteros sin signo y devuelve un entero sin signo.

7.33. *a*) `f` acepta dos argumentos en coma flotante y devuelve un valor en coma flotante.
b) `f` acepta un entero largo y devuelve un entero largo.
c) `f` acepta un entero y no devuelve nada.
d) `f` no acepta nada y devuelve un carácter.

7.34. *a*) `y = formula(x);`
b) `escribe(a, b);`

7.35. *a*) `int muestra(void);`
b) `float raiz(int a, int b);`
c) `char convertir(char c)`
d) `char transferir(long i)`
e) `long inversa(char c)`
f) `double procesar(int i, float a, float b)`
g) `void valor(double x, double y, short i)`

7.36. *a*) `int func1(int a, int b);`
b) `double func1(double a, double b);`
c) `long int func1(int a, float b);`
d)
```
double func1(double a, double b);
double func2(double a, double b);
```

7.37. *a*) `1   4   9   16   25`

b)
```
#include <stdio.h>

int func1(int cont);

main()
{
  int cont;

  for (cont = 1; cont <= 5; ++cont)
      printf("%d  ", func1(cont));
}

int func1(int x)
{
    return(x * x);
}
```

c) 55

d) 30

7.38. *a*) $y = x_n + \sum_{i=1}^{n-1} x_i$ o

$y_1 = x_1$, e $y_n = x_n + y_{n-1}$ para $n > 1$

b) $y = (-1)^n x_n / n! + \sum_{i=0}^{n-1} (-1)^i x_i / i!$ o

$y_0 = 1$, e $y_n = (-1)^n x_n / n! + y_{n-1}$ para $n > 0$

c) $p = f_t * \prod_{j=1}^{t-1} f_j$ o

$p_1 = f_1$, y $p_t = f_t * p_{t-1}$ para $n > 1$

Capítulo 8

8.25. *a*) `1   2   3   4   5`
b) `1   3   6   10   15`
c) `6   15   28   45   66`

8.26. *a*) `extern float resolver(float a, float b)`

Tenga en cuenta que `extern` puede ser omitido; es decir, la primera línea se puede escribir así:

`float resolver(float a, float b)`

b) `static float resolver(float a, float b)`

8.27. *a*) *Primer archivo*:

```
extern double func1(double a, double b); /* añadida */
main()
{
  double x, y, z;
  . . . . .
  z = func1(x, y);
  . . . . .
}
```

Segundo archivo:

```
double func1(double a, double b)
{
  . . . . .
}
```

b) *Primer archivo*:

```
extern double func1(double x, double y); /* añadida */
extern double func2(double x, double y); /* añadida */
main()
{
  double x, y, z;
  . . . . .
  z = func1(x, y);
  . . . . .
}
```

Segundo archivo:

```
double func1(double a, double b)
{
  double c;

  c = func2(a, b);
  . . . . .
}
static double func2(double a, double b)
{
  . . . . .
}
```

8.28. *a*) `4 6 9 13 18`

b) `100 196 80 184 60 164 40 136 20 100`

c) 104 116 136 136 100
d) 101 102 106 124 200
e) 6 11 16 21 26
f) 6 11 16 21 26
g) 9 25 57 121 249
h) Este programa devolverá el número de caracteres de una línea de texto introducida por teclado. El carácter fin de línea no se incluirá en la suma.

Capítulo 9

9.27. *a*) `nombre` es un array unidimensional de caracteres de 30 elementos.
b) `c` es un array unidimensional de 6 elementos en coma flotante.
c) `a` es un array unidimensional de 50 elementos enteros.
d) `parametros` es un array bidimensional de 25 elementos enteros (5 filas, 5 columnas).
e) `memo` es un array bidimensional de caracteres de 8712 elementos (66 filas y 132 columnas).
f) `cuentas` es un array tridimensional de 80.000 elementos de doble precisión (50 páginas, 20 filas y 80 columnas).

9.28. *a*) `c` es un array unidimensional de 8 elementos en coma flotante.

```
c[0] = 2.     c[1] = 5.    c[2] = 3.  c[3] = -4.
c[4] = 12.    c[5] = 12.   c[6] = 0.  c[7] = 8.
```

b) `c` es un array unidimensional de 8 elementos en coma flotante.

```
c[0] = 2.      c[1] = 5.      c[2] = 3.      c[3] = -4.
c[4] = 0.      c[5] = 0.      c[6] = 0.      c[7] = 0.
```

c) `z` es un array unidimensional de 12 elementos enteros.

`z[2] = 8` `z[5] = 6` El resto de los elementos tienen asignados ceros.

d) `indicador` es un array unidimensional de caracteres de 9 elementos.

```
indicador[0] = 'V'   indicador[1] = 'E'
indicador[2] = 'R'   indicador[3] = 'D'
indicador[4] = 'A'   indicador[5] = 'D'
indicador[6] = 'G'   indicador[7] = 'R'
indicador[8] = 'O'
```

e) `indicador` es un array unidimensional de caracteres de 10 elementos.

```
indicador[0] = 'V'   indicador[1] = 'F'
indicador[2] = 'R'   indicador[3] = 'D'
indicador[4] = 'A'   indicador[5] = 'D'
indicador[6] = 'G'   indicador[7] = 'R'
indicador[8] = 'O'   indicador[9] tiene asignado cero
```

f) `indicador` es un array unidimensional de caracteres de 5 elementos.

```
indicador[0] = 'V'   indicador[1] = 'F'
indicador[2] = 'R'   indicador[3] = 'D'
indicador[4] = 'A'   indicador[5] = 'D'
indicador[6] = 'G'   indicador[7] = 'R'
indicador[8] = 'O'   indicador[9] = '\O'
```

g) `indicador` es un array unidimensional de caracteres de 6 elementos.

```
indicador[0] = 'F'      indicador[1] = 'A'
indicador[2] = 'L'      indicador[3] = 'S'
indicador[4] = 'O'      indicador[5] = '\0'
```

h) p es un array unidimensional de 2×4 enteros.

```
p[0][0] = 1   p[0][1] = 3   p[0][2] = 5   p[0][3] = 7
p[1][0] = 0   p[1][1] = 0   p[1][2] = 0   p[1][3] = 0
```

i) p es un array bidimensional de 2×4 enteros.

```
p[0][0] = 1   p[0][1] = 1   p[0][2] = 3   p[0][3] = 3
p[1][0] = 5   p[1][1] = 5   p[1][2] = 7   p[1][3] = 7
```

j) p es un array bidimensional de 2×4 enteros.

```
p[0][0] = 1   p[0][1] = 3   p[0][2] = 5   p[0][3] = 7
p[1][0] = 2   p[1][1] = 4   p[1][2] = 6   p[1][3] = 8
```

k) p es un array bidimensional de 2×4 enteros.

```
p[0][0] = 1   p[0][1] = 3   p[0][2] = 0   p[0][3] = 0
p[1][0] = 5   p[1][1] = 7   p[1][2] = 0   p[1][3] = 0
```

l) c es un array tridimensional de 2×3×4 enteros.

```
c[0][0][0] = 1  c[0][0][1] = 2  c[0][0][2] = 3  c[0][0][3] = 0
c[0][1][0] = 4  c[0][1][1] = 5  c[0][1][2] = 0  c[0][1][3] = 0
c[0][2][0] = 6  c[0][2][1] = 7  c[0][2][2] = 8  c[0][2][3] = 9
c[1][0][0] = 10 c[1][0][1] = 11 c[1][0][2] = 0  c[1][0][3] = 0
c[1][1][0] = 0  c[1][1][1] = 0  c[1][1][2] = 0  c[1][1][3] = 0
c[1][2][0] = 12 c[1][2][1] = 13 c[1][2][2] = 14 c[1][2][3] = 0
```

m) `colores` es un array bidimensional de 3×6 caracteres.

```
colores[0][0] = 'R' colores[0][1] = 'O' colores[0][2] = 'J'
colores[0][3] = 'O' colores[0][4] = 0   colores[0][5] = 0
colores[1][0] = 'V' colores[1][1] = 'E' colores[1][2] = 'R'
colores[1][3] = 'D' colores[1][4] = 'E' colores[1][5] = 0
colores[2][0] = 'A' colores[2][1] = 'Z' colores[2][2] = 'U'
colores[2][3] = 'L' colores[2][4] = 0   colores[2][5] = 0
```

9.29. *a*) `int c[12] = {1, 4, 7, 10, 13, 16, 19, 22, 25, 28, 31, 34};`

b) `char punto[] = "NORTE";`

c) `char letras[4] = {'N', 'S', 'E', 'O'};`

d)
```
float constantes[6] = {0.005, -0.032, 1e-6, 0.167, -0.3e8,
                       0.015};
```

e)
```
int n[3][4] = {10, 12, 14, 16, 20, 22, 24, 26, 30, 32, 34, 36};
```

Otra forma de asignar valores iniciales es la siguiente:

```
int n[3][4] = {
                  {10, 12, 14, 16},
                  {20, 22, 24, 26},
                  {30, 32, 34, 36}
              };
```

f)
```
int n[3][4] = {10, 12, 14, 0, 0, 20, 22, 0, 0, 30, 32, 0};
```

o

```
int n[3][4] = {
                  {10, 12, 14},
                  {0, 20, 22},
                  {0, 30, 32}
              };
```

g)
```
int n[3][4] = {10, 12, 14, 16, 20, 22};
```

9.30. *a*)
```
float muestra(float a, float b, int jstar[]);

main()
{
  float a, b, c;
  int jstar[20];

  . . . . .

  x = muestra(a, b, jstar);

  . . . . .
}

float muestra(float a, float b, int jstar[])
{
  . . . . .
}
```

b)
```
float muestra(int n, char c, double valores[]);

main()
{
  int n;
  char c;
  float x;
  double valores[50];

  . . . . .

  x = muestra(n, c, valores);

  . . . . .
}
```

```
float muestra(int n, char c, double valores[])
{
 . . . . .
}
```

c)
```
float muestra(char texto[][80]);

main()
{
  float x;
  char texto[12][80];

  . . . . .

  x = muestra(texto);

  . . . . .

}

float muestra(char texto[][80])
{
  . . . . .
}
```

d)
```
float muestra(char mensaje[], float cuentas[][100]);

main()
{
  float x;
  char mensaje[40];
  float cuentas[50][100];

  . . . . .

  x = muestra(mensaje, cuentas);

  . . . . .

}

float muestra(char mensaje[], float cuentas[][100])
{
  . . . . .
}
```

9.31. *a*) 20 (suma de los elementos de arrays cuyos valores son pares)
b) 25 (suma de los elementos pares del array)
c) No funcionará (los arrays automáticos no pueden inicializarse)
d) 25 (suma de los elementos de valor impar de un array externo)
e) 1 (el valor más pequeño)

f) 1 5 9 (valor más pequeño dentro de cada fila)
g) 9 10 11 12 (valor más grande dentro de cada columna)
h) 0 2 2 4
4 6 6 8
8 10 10 12 (si el valor de un elemento es impar, reducir su valor en 1; mostrar a continuación el array completo)
i) `PPoorrmmrree    uudd  ee  uu  iieettdd!!` (saltar los elementos con índice impar del array; imprimir cada elemento con índice par dos veces)

Capítulo 10

10.44. *a*) `px` es un puntero a una cantidad entera.
b) `a` y `b` son variables en coma flotante; `pa` y `pb` son punteros a cantidades en coma flotante (aunque no necesariamente a `a` y a `b`).
c) `a` es una variable en coma flotante de valor inicial `-0.167`; `pa` es un puntero a una cantidad en coma flotante; la dirección de `a` se asigna a `pa` como valor inicial.
d) `c1`, `c2` y `c3` son variables de tipo carácter; `pc1`, `pc2` y `pc3` son punteros a caracteres; la dirección de `c1` se asigna a `pc3`.
e) `func` es una función que acepta tres argumentos y devuelve una cantidad de doble precisión. Los dos primeros argumentos son punteros a cantidades de doble precisión; el tercer argumento es un puntero a una cantidad entera.
f) `func` es una función que acepta tres argumentos y devuelve un puntero a una cantidad de doble precisión. Los dos primeros argumentos son punteros a cantidades de doble precisión; el tercer argumento es un puntero a una cantidad entera.
g) `a` es un puntero a un grupo de arrays unidimensionales, de doble precisión, contiguos; esto es equivalente a `double a[][12];`
h) `a` es un array unidimensional de punteros a cantidades de doble precisión (equivalente a un array bidimensional de cantidades de doble precisión).
i) `a` es un array unidimensional de punteros a caracteres simples o cadenas de caracteres (equivalente a un array bidimensional de caracteres).
j) `d` es un array unidimensional de punteros a las cadenas `"norte"`, `"sur"`, `"este"` y `"oeste"`.
k) `p` es un puntero a un grupo de arrays bidimensionales, de enteros largos, contiguos; equivalente a `p[][10][20];`
l) `p` es un array bidimensional de punteros a cantidades enteras largas (equivalente a un array tridimensional de enteros largos).
m) `muestra` es una función que acepta un argumento que es una función y devuelve un carácter. La función pasada como argumento acepta dos argumentos carácter y devuelve una cantidad entera.
n) `pf` es un puntero a una función que no acepta argumentos pero que devuelve una cantidad entera.
o) `pf` es un puntero a una función que acepta dos argumentos carácter y devuelve una cantidad entera.
p) `pf` es un puntero a una función que acepta dos punteros a caracteres como argumentos y devuelve una cantidad entera.

10.45. *a*)
```
int i, j;
int *pi = &i;
int *pj = &j;
```

b) `float *pf;`
`double *pd;`
c) `long *func(int a, int b);`
d) `long func(int *a, int *b);`
e) `float *x;`
f) `float (*x)[30];` o `float *x[15]`
g) `char *color[3] = {"rojo", "verde", "azul"};`
h) `char *func(int (*pf)(int a));`
i) `float (*pf)(int a, int b, int c);`
j) `float *(*pf)(int *a, int *b, int *c);`

10.46. *a*) `F8D` *c*) `'B'` *e*) `F8C` *g*) `'C'`
b) `F8D` *d*) `'C'` *f*) `F8C`

10.47. *a*) `F9C`
b) `F9E`
c) `F9E`
d) `30` (observe que esto cambia el valor de `j`)
e) `35`
f) `F9E`
g) `(i + j) = 35 + 30 = 65`
h) `FA2`
i) `67`
j) no especificado

10.48. *a*) `1130` *d*) `1130` *g*) `1134`
b) `1134` *e*) `0.002` *h*) `0.003`
c) `1138` *f*) `&(*pa) = pa = 1130` *i*) `0.003`

10.49. *a*) `80` *c*) `a=88 b=89`
b) `81` *d*) `a=80 b=81`

10.50. *a*) Un puntero a un entero.
b) No se devuelve nada.
c) Un puntero a una cantidad entera.
d) Calcula la suma de los elementos de `p` (`p` es un array de cinco elementos enteros).
e) `suma=150`

10.51. *a*) Un puntero a un entero.
b) No se devuelve nada.
c) Los dos últimos elementos de un array de cinco elementos enteros.
d) Calcula la suma de los dos últimos elementos de un array de cinco elementos enteros.
e) `suma=90`

10.52. *a*) Un puntero a un entero.
b) Un puntero a un entero.
c) La dirección del elemento de `p` cuyo valor es el mayor (`p` es realmente un array de cinco elementos enteros).

d) Determinar el mayor valor de los elementos de p.
e) max=50

10.53. *a*) Dirección de x[0] *d*) 12 (esto es, 10 + 2)
b) Dirección de x[2] *e*) 30 (éste es el valor de x[2])
c) 10

10.54. *a*) Dirección de tabla[0][0]
b) Dirección de fila 1 (la segunda fila) de tabla
c) Dirección de tabla[1][0]
d) Dirección de tabla[1][1]
e) Dirección de tabla[0][1]
f) 2.2
g) 1.2
h) 2.1
i) 2.2 (esto es, 1.2 + 1)

10.55. *a*) Dirección de color[0] (comienzo de la primera cadena)
b) Dirección de color[2] (comienzo de la tercera cadena)
c) "rojo"
d) "azul"
e) Los dos refieren al mismo elemento del array (puntero a "amarillo")

10.56. *a*) a y b son variables ordinarias en coma flotante. uno, dos y tres son funciones, cada una de las cuales devuelve una cantidad en coma flotante. uno y dos aceptan, cada una, dos cantidades en coma flotante como argumentos. tres acepta una función como argumento; la función argumento aceptará dos cantidades en coma flotante como sus propios argumentos y devolverá una cantidad en coma flotante. (Observe que tanto uno como dos pueden aparecer como argumentos de tres.)
b) uno y dos son definiciones de funciones convencionales. Cada una acepta dos cantidades en coma flotante y devuelve una cantidad en coma flotante que se calcula dentro de la función.
c) tres acepta un puntero a una función como argumento. La función argumento acepta dos cantidades en coma flotante y devuelve una cantidad en coma flotante. Dentro de tres se accede a la función argumento y el resultado calculado se asigna a c. El valor de c es luego devuelto a main.
d) Cada vez que se accede a tres se pasa una función diferente. Por tanto, el valor devuelto por tres se calculará de modo distinto cada vez que se acceda a dicha función.

10.57. *a*) a y b son punteros a cantidades en coma flotante. uno, dos y tres son funciones; uno y dos devuelven, cada una, una cantidad en coma flotante y tres devuelve un puntero a una cantidad en coma flotante. uno y dos aceptan, cada una, dos punteros a cantidades en coma flotante como argumentos. tres acepta una función como argumento; la función argumento aceptará dos punteros a cantidades en coma flotante como sus propios argumentos y devolverá una cantidad en coma flotante. (Observe que tanto uno como dos pueden aparecer como argumentos de tres.)
b) uno y dos son definiciones de funciones convencionales. Cada una acepta dos punteros a cantidades en coma flotante y devuelve una cantidad en coma flotante que se calcula dentro de la función.

c) `tres` acepta un puntero a una función como argumento. La función argumento acepta dos punteros a cantidades en coma flotante y devuelve una cantidad en coma flotante. Dentro de `tres` se accede a la función argumento y el resultado calculado se asigna a `c`. La dirección de `c` es luego devuelta a `main`.

d) Cada vez que se accede a `tres` se pasa una función diferente. Por tanto, el valor cuya dirección es devuelta por `tres` se calculará de modo distinto cada vez que se acceda a dicha función.

e) En este esquema `uno` y `dos` aceptan punteros como argumentos, mientras que en el esquema anterior aceptaban variables ordinarias en coma flotante como argumentos. También en este esquema `tres` devuelve un puntero, mientras que en el anterior devolvía una cantidad en coma flotante ordinaria.

10.58. *a*) `x` es un puntero a una función que acepta un puntero a una cantidad entera como argumento y devuelve una cantidad en coma flotante.

b) `x` es una función que acepta un puntero a una cantidad entera como argumento y devuelve un puntero a un array de `20` elementos en coma flotante.

c) `x` es una función que acepta un puntero a un array de enteros como argumento y devuelve una cantidad en coma flotante.

d) `x` es una función que acepta un array de punteros a enteros como argumento y devuelve una cantidad en coma flotante.

e) `x` es una función que acepta un array de enteros como argumento y devuelve un puntero a una cantidad en coma flotante.

f) `x` es una función que acepta un puntero a un array de enteros como argumento y devuelve un puntero a una cantidad en coma flotante.

g) `x` es una función que acepta un array de punteros a cantidades enteras como argumento y devuelve un puntero a una cantidad en coma flotante.

h) `x` es un puntero a una función que acepta un puntero a un array de enteros como argumento y devuelve una cantidad en coma flotante.

i) `x` es un puntero a una función que acepta un array de punteros a cantidades enteras como argumento y devuelve un puntero a una cantidad en coma flotante.

j) `x` es un array de punteros a funciones de `20` elementos; cada función acepta un entero como argumento y devuelve una cantidad en coma flotante.

k) `x` es un array de punteros a funciones de `20` elementos; cada función acepta un puntero a una cantidad entera como argumento y devuelve un puntero a una cantidad en coma flotante.

10.59.
a) `char (*p(int *a))[6];`
b) `char p(int (*a)[]);`
c) `char p(int *a[]);`
d) `char *p(int a[]);`
e) `char *p(int (*a)[]);`
f) `char *p(int *a[]);`
g) `char (*p)(int (*a)[]);`
h) `char *(*p)(int (*a)[]);`
i) `char *(*p)(int *a[]);`
j) `double (*f[12])(double a, double b);`
k) `double *(*f[12])(double a, double b);`
l) `double *(*f[12])(double *a, double *b);`

Capítulo 11

11.34.
```
struct  complejo  {
   float real;
   float imaginario;
};
```

11.35.
```
struct  complejo  x1, x2, x3;
```

11.36.
```
struct  complejo  {
     float real;
     float imaginario;
}  x1, x2, x3;
```

Es opcional en esta situación incluir la marca (`complejo`).

11.37.
```
struct  complejo  x = {1.3, -2.2};
```

Recordar que `x` puede ser tanto `static` como `external`.

11.38.
```
struct  complejo  *px;
```

Los miembros de la estructura son `px->real` y `px->imaginario`.

11.39.
```
struct  complejo  cx[100];
```

11.40.
```
struct  complejo  {
     float real;
     float imaginario;
}  cx[100];
```

Es opcional en esta situación incluir la marca (`complejo`).

11.41. Los miembros de la estructura son `cx[17].real` y `cx[17].imaginario`.

11.42.
```
typedef  struct  {
     int ganados;
     int perdidos;
     float porcentaje;
}  registro;
```

11.43.
```
typedef  struct  {
     char nombre[40];
     registro posicion;
}  equipo;
```

donde el tipo de estructura `registro` está definido en el Problema 11.42.

11.44.
```
equipo  t;
```

Todos los miembros de la estructura son `t.nombre`, `t.posicion.ganados`, `t.posicion.perdidos` y `t.posicion.porcentaje`. Los caracteres que constituyen `t.nombre` pueden ser accedidos individualmente; por ejemplo, `t.nombre[0]`, `t.nombre[1]`, `t.nombre[2]`, . . ., etc.

11.45. `equipo t ={"Osos de Chicago", 14, 2, 87.5};`

11.46. `printf("%d\n", sizeof t);`

o

`printf("%d\n", sizeof (equipo));`

11.47. `equipo *pt;`

Todos los miembros de la estructura son `pt->nombre`, `pt->posicion.ganados`, `pt->posicion.perdidos` y `pt->posicion.porcentaje`. Los caracteres que componen `pt->nombre` pueden ser accedidos individualmente; por ejemplo, `pt->nombre[0]`, etc.

11.48. `equipo liga[48];`

Los elementos individuales son `liga[4].nombre` y `liga[4].posicion.porcentaje`.

11.49.

```
struct equipo {
    char nombre[40];
    registro posicion;
    struct equipo *sig;
};
```

11.50. Se dan dos soluciones y ambas son correctas.

a)
```
struct equipo *pt;

pt = (struct equipo *) malloc(sizeof(struct equipo));
```

b)
```
typedef struct equipo ciudad;
ciudad *pt;

pt = (ciudad *) malloc(sizeof(ciudad));
```

11.51. Se dan dos soluciones y ambas son correctas.

a)
```
struct hms {
     int hora;
     int minuto;
     int segundo;
};
union {
    struct hms local;
    struct hms casa;
} *hora;
```

b)
```
typedef struct {
     int hora;
     int minuto;
     int segundo;
} hms;

union {
    hms local;
    hms casa;
} *hora;
```

11.52. Se dan dos soluciones y ambas son correctas.

a)
```
union res  {
    int eres;
    float fres;
    double dres;
};

struct  {
    union res respuesta;
    char indicador;
    int a;
    int b;
}  x, y;
```

b)
```
typedef  union  {
    int eres;
    float fres;
    double dres;
}  res;

struct  {
    res respuesta;
    char indicador;
    int a;
    int b;
}  x, y;
```

11.53.
```
union res  {
     int eres;
     float fres;
     double dres;
};

struct  muestra  {
     union res respuesta;
     char indicador;
     int a;
     int b;
};
struct  muestra  v = {14, 'e', -2, 5};
```

11.54.
```
union res  {
   int eres;
   float fres;
   double dres;
};
struct  muestra  {
   union res respuesta;
   char indicador;
   int a;
   int b;
   struct muestra *sig;
};
```

```
typedef struct muestra tipo_estructura;
tipo_estructura x, *px = &x;
```

11.55. *a*)
```
rojo verde azul
cian magenta amarillo
rojo verde azul
```

La variable de estructura muestra es pasada a func por valor. Por tanto, las reasignaciones dentro de func no son reconocidas dentro de main.

b)
```
rojo verde azul
cian magenta amarillo
cian magenta amarillo
```

La variable de estructura muestra es pasada a func por referencia. (En realidad se pasa a func un puntero al comienzo de muestra.) Por tanto, las reasignaciones dentro de func son reconocidas dentro de main.

c)
```
rojo verde azul
cian magenta amarillo
cian magenta amarillo
```

La variable de estructura muestra es pasada a func por valor, como en *a*). Ahora, sin embargo, se devuelve a main la variable de estructura modificada.

11.56.
```
8
100 0.000000 -0.000000
0 0.500000 -0.000000
-25098 391364288.000000 0.016667
```

La primera línea representa el tamaño de la unión (8 bytes para acomodar un número de doble precisión). En la segunda línea sólo tiene sentido el primer valor (100). En la tercera línea sólo tiene sentido el segundo valor (0.500000). Y en la última línea sólo tiene sentido el último valor (0.016667).

11.57. *a*)
```
200 0.500012
0 0.500000
```

La variable de unión u se pasa a func por valor. Por tanto, no se reconoce dentro de main la reasignación dentro de func. Observe que sólo el primer valor tiene sentido en la primera línea de salida y sólo el segundo en la última línea.

b)
```
-26214 -0.300000
0 0.500000
```

La variable de unión u se pasa de nuevo a func por valor. Luego, dentro de main no se reconoce la reasignación efectuada dentro de func. El primer valor de cada línea no tiene sentido.

c)
```
-26214 -0.300000
-26214 -0.300000
```

La variable de unión u se pasa a func por valor, pero a main se devuelve la variable de unión modificada. Por tanto, la reasignación efectuada dentro de func será reconocida dentro de main. El primer valor de cada línea no tiene sentido.

Capítulo 12

12.21.

```
#include <stdio.h>

FILE *puntr;

puntr = fopen("estudian.dat", "w");
```

12.22.

```
#include <stdio.h>

FILE *puntr;

puntr = fopen("estudian.dat", "r+");
fclose(puntr);
```

12.23.

```
#include <stdio.h>

FILE *puntr;

puntr = fopen("muestra.dat", "w+");

fclose(puntr);
```

12.24.

```
#include <stdio.h>

FILE *puntr;

puntr = fopen("muestra.dat", "r+");

fclose(puntr);
```

12.25.

```
#include <stdio.h>
#define  NULL  0

FILE *puntr;

puntr = fopen("muestra.dat", "r+");
if (puntr == NULL)
   printf("\nERROR - No se puede abrir el archivo designado\n");

fclose(puntr);
```

A menudo se combinan las instrucciones `fopen` e `if`, por ejemplo:

```
if ((puntr = fopen("muestra.dat", "r+")) == NULL)
   printf("\nERROR - No se puede abrir el archivo designado\n");
```

12.26.

```
printf("Introducir valores para a, b y c: ");
scanf("%d %f %c", &a, &b, &c);
fprintf(fpt, "%d %.2f %c", a, b, c);
```

Se pueden incluir, si se desea, los caracteres de nueva línea (`\n`) en la cadena de control de `fprintf`.

12.27.

```
fscanf(fpt, "%d %f %c", &a, &b, &c);
printf("a = %d   b = %f   c = %c", a, b, c);
```

12.28. *a*) `fscanf(pt1, "%d %f %c", &a, &b, &c);`

b)
```
printf("a = %d   Nuevo valor: ", a);
scanf("%d", &a);
printf("b = %f   Nuevo valor: ", b);
scanf("%f", &b);
printf("c = %c   Nuevo valor: ", c);
scanf("%c", &c);
```

c) `fprintf(pt2, "%d %.2f %c", a, b, c);`

Se pueden incluir, si se desea, los caracteres de nueva línea (`\n`) en la cadena de control de `fprintf`.

12.29. *a*) `fscanf(pt1, "%s", nombre);`

b) `printf("Nombre: %s\n", nombre);`

c)
```
printf("Nuevo nombre: ");
scanf(" %[^\n]", nombre);
```

d) `fprintf(pt2, "%s", nombre);`

He aquí otra solución:

a) `fgets(nombre, 20, pt1);`

b) `printf("Nombre: %s\n", nombre);`

c)
```
puts("Nuevo nombre: ");
gets(nombre);
```

d) `fputs(nombre, pt2);`

12.30. *a*)
```
fscanf(pt1, "%s", valores.nombre);
printf("%s", valores.nombre);
```

b)
```
printf("a = ");
scanf("%d", &valores.a);
printf("b = ");
scanf("%f", &valores.b);
printf("c = ");
scanf("%c", &valores.c);
```

c)
```
fprintf(pt2, "%s %d %f %c", valores.nombre, valores.a,
                            valores.b, valores.c);
```

o

```
fprintf(pt2, "%s\n%d\n%f\n%c\n", valores.nombre,valores.a,
                                 valores.b, valores.c);
```

o

```
fprintf(pt2, "%s\n", valores.nombre);
fprintf(pt2, "%d\n", valores.a);
fprintf(pt2, "%f\n", valores.b);
fprintf(pt2, "%c\n", valores.c);
```

12.31. *a*)
```
fread(&valores, sizeof valores, 1, pt1);
printf("%s", valores.nombre);
```

b)
```
printf("a = ");
scanf("%d", &valores.a);
printf("b = ");
scanf("%f", &valores.b);
printf("c = ");
scanf("%c", &valores.c);
```
c)
```
fwrite(&valores, sizeof valores, 1, pt2);
```

Capítulo 13

13.36.
```
register  unsigned  u, v;
```

13.37.
```
int u = 1, v = 2;
register int x = 3, y = 4;
```

13.38.
```
unsigned *func(register unsigned *pt1);    /* prototipo de función */

main()
{
  register unsigned *pt1;             /* declaración de puntero */
  unsigned *pt2;                      /* declaración de puntero */
  . . . . .
  pt2 = func(pt1);
  . . . . .
}

unsigned *func(register unsigned *pt1)    /* definición de función */
{
  unsigned *pt2;
  . . . . .
  pt2 = . . . . .;
  . . . . .
  return(pt2);
}
```

13.39. Patrón de bits correspondiente a a: `1010 0010 1100 0011`

```
a) 5d3c   0101 1101 0011 1100
b) 2202   0010 0010 0000 0010
c) 9dc5   1001 1101 1100 0101
d) bfc7   1011 1111 1100 0111
e) 80c1   1000 0000 1100 0001
f) 623a   0110 0010 0011 1010
g) e2fb   1110 0010 1111 1011
h) 1458   0001 0100 0101 1000
```

i)	5860	0101 1000 0110 0000	
j)	0	0000 0000 0000 0000	(válido para cualquier valor de a)
k)	ffff	1111 1111 1111 1111	(válido para cualquier valor de a)
l)	ffff	1111 1111 1111 1111	(válido para cualquier valor de a)
m)	a000	1010 0000 0000 0000	
n)	c100	1100 0001 0000 0000	
o)	a0c3	1010 0000 1100 0011	
p)	5bc3	0101 1011 1100 0011	
q)	3a00	0011 1010 0000 0000	
r)	5b3c	0101 1011 0011 1100	
s)	fbc3	1111 1011 1100 0011	
t)	fb00	1111 1011 0000 0000	
u)	fbff	1111 1011 1111 1111	

13.40. *a*) `a &= 0x3f06` *b*) `a^= 0x3f06` *c*) `a |= ~0x3f06` *d*) `a >>= 3` *e*) `a <<= 5` *f*) `a ^= ~a` *g*) `a &= ~(0x3f06 << 8)`

13.41. *a*) `v & 0xaaaa` o `v & ~0x5555` *b*) `c & 0x7f` *c*) `v | 0x5555` *d*) `v ^= 0x42`

13.42. *a*) Observe que v representa un número positivo, ya que el bit del extremo izquierdo es 0 (el valor decimal equivalente es 13980). Por tanto, los bits desocupados como resultado de ambas operaciones de desplazamiento se rellenarán con ceros. Los valores resultantes son

i) 0x69c0 ii) 0x369

b) Ahora v representa un número negativo, ya que el bit del extremo izquierdo es 1 (el valor decimal equivalente es -15511). Por tanto, los bits desocupados en la operación de desplazamiento a la izquierda se rellenarán con ceros, pero los bits desocupados en la operación de desplazamiento a la derecha se rellenarán con unos. Los valores resultantes son

i) 0x3690 ii) 0xfc36

13.43. Cada estructura define varios campos de bits.

a) u consta de 3 bits, v de 1 bit, w de 7 bits y x de 5 bits. El total de bits es 16. Por tanto, todos los campos de bits ocuparán una palabra.

b) Los campos de bits individuales son los mismos que en la parte *a*). Sin embargo, ahora a cada campo de bits se le asigna un valor inicial. Observe que cada valor es lo suficientemente pequeño para acomodarse dentro de su correspondiente campo de bits (por ejemplo, 2 requiere dos bits, 1 un bit, 16 cinco bits y 8 cuatro bits).

c) u, v y w son de 7 bits de tamaño cada uno. Se necesitarán dos palabras de memoria. u y v se acomodarán en una palabra, pero w será forzado al comienzo de la siguiente palabra.

d) u, v y w son de 7 bits de tamaño cada uno. Se necesitarán dos palabras. u se colocará en la primera palabra, seguido por 9 bits vacíos. v y w se acomodarán en la segunda palabra, separados por dos bits vacíos.

e) u, v y w son de 7 bits de tamaño cada uno. Se utilizarán tres palabras para almacenar estos campos de bits. u se colocará en la primera palabra, v será forzado al comienzo de la segunda palabra, y w será forzado al comienzo de la tercera palabra. Cada campo de bits va seguido por 9 bits vacíos.

13.44. *a*)
```
struct campos {
    unsigned a : 6;
    unsigned b : 4;
    unsigned c : 6;
};
```

b)
```
static struct campos v = {3, 5, 7};
```

o

```
static struct {
    unsigned a : 6;
    unsigned b : 4;
    unsigned c : 6;
} v = {3, 5, 7};
```

Cada valor puede acomodarse en un campo de tres bits.

c) Los campo de 6 bits pueden acomodar cualquier valor hasta 63, ya que

$63 = 2^6 - 1 = 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$

Los campos de 4 bits pueden acomodar cualquier valor hasta 15, ya que $15 = 2^4 - 1 = 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$

d)
```
static struct {
    unsigned a : 8;
    unsigned b : 6;
    unsigned c : 5;
};
```

a y b se almacenarán en una palabra de 16 bits, y c se almacenará en una segunda palabra de 16 bits.

e)
```
static struct {
    unsigned a : 8;
    unsigned   : 2;
    unsigned b : 6;
    unsigned c : 5;
};
```

f)
```
static struct {
    unsigned a : 8;
    unsigned   : 0;
    unsigned b : 6;
    unsigned   : 2;
    unsigned c : 5;
};
```

Capítulo 14

14.24.
```
enum indicadores {primero, segundo, tercero, cuarto, quinto};
```

14.25.
```
enum indicadores suceso;
```

o

```
enum {primero, segundo, tercero, cuarto, quinto} suceso;
```

14.26.
```
enum  {do = 1, re, mi, fa, sol, la, si}  soprano, bajo;
```

14.27.
```
enum  dinero  {penique = 1, niquel = 5, dime = 10,
               cuarto = 25, medio = 50, dolar = 100};
```

14.28.
```
enum  dinero  moneda = dime;
```

o

```
enum  {penique = 1, niquel = 5, dime = 10,cuarto = 25,
       medio = 50, dolar = 100}  moneda = dime;
```

14.29.
```
norte = 2
sur = 3
este = 1
oeste = 2
```

14.30.
```
mover_1 = 3
mover_2 = 2
```

14.31. Esta instrucción `switch` calcula los tantos acumulados, utilizando las reglas que dependen de los valores asignados a la variable de enumeración `mover`. Las reglas son las siguientes: si `mover = norte` sumar 10 puntos a `tantos`; si `mover = sur` sumar 20 puntos a `tantos`; si `mover = este` sumar 30 puntos a `tantos`; si `mover = oeste` sumar 40 puntos a `tantos`. Se muestra un mensaje de error si a `mover` se le asigna un valor diferente de `norte`, `sur`, `este` u `oeste`.

14.32. *a*) `argc=3, argv[0]=demo, argv[1]=depurar, argv[2]=rapido`
b) `argc=2, argv[0]=demo` y `argv[1]=depurar rapido`

14.33. Este programa leerá una línea de texto y la mostrará en mayúsculas o minúsculas, dependiendo del segundo parámetro de la línea de órdenes. Este parámetro debe ser o `mayuscula` o `minuscula`. Si no es ninguno de los dos, se genera un mensaje de error y no se muestra el texto.

14.34.
```
transfer.exe  datos.ant  datos.nue
```

o, con algunos compiladores,

```
transfer  datos.ant  datos.nue
```

14.35.
a) `#define  PI  3.1415927`
b) `#define  AREA  PI * radio * radio`
c) `#define  AREA(radio)  PI * radio * radio`
d) `#define  CIRCUNFERENCIA  2 * PI * radio`
e) `#define  CIRCUNFERENCIA (radio)  2 * PI * radio`
f)
```
#define  interes  {                            \
                     i = 0.01 * r;             \
                     f = p * pow((1 + i), n);  \
                  }
```

Se supone que las variables `i`, `r`, `f`, `p` y `n` han sido declaradas como variables de doble precisión.

g)
```
#define  interes(p, r, n)  {              \
             i = 0.01 * r;                \
             f = p * pow((1 + i), n);     \
         }
```

h)
```
#define  max  (a >= b)  ? a : b
```

o

```
#define  max  (((a) >= (b))  ? (a) : (b))
```

La segunda versión minimizará la probabilidad de efectos laterales no deseables.

i)
```
#define  max(a, b)  (a >= b)  ? a : b
```

o

```
#define  max(a, b)  (((a) >= (b))  ? (a) : (b))
```

14.36. *a*) Si la constante simbólica `INDICADOR` no ha sido definida previamente, se define `INDICADOR` para representar el valor `1`.

b) Si la constante simbólica `PASCAL` ha sido definida previamente, se definen las constantes simbólicas `BEGIN` y `END` para representar los símbolos `{` y `}`, respectivamente.

c) Si la constante simbólica `CELSIUS` ha sido definida previamente, se define la macro `temperatura(t)` para representar la expresión `0.5555555 * (t - 32)`; en otro caso, se define `temperatura` de modo que represente la expresión `1.8 * t + 32`.

d) Si la constante simbólica `DEPURAR` no ha sido definida previamente, se define la macro `salida` como

```
printf("x = %f\n", x)
```

En otro caso, si la constante simbólica `NIVEL` tiene un valor de `1`, se define `salida` como

```
printf("i = %d   y = %f\n", i, y[i])
```

Y si `NIVEL` no tiene el valor `1`, se define `salida` como la macro multilínea

```
for (cont = 1; cont <= n; ++cont)       \
     printf("i = %d   y = %f\n", i, y[i])
```

(Se supone que las variables `x`, `i`, `y`, `cont` y `n` han sido declaradas adecuadamente.)

e) «Anula» la definición de la constante simbólica `DEPURAR` si ésta ha sido definida previamente.

f) Este problema ilustra el uso del operador «de cadena» (#). Si la constante simbólica `COMPROBACION_ERROR` ha sido definida previamente, entonces se define la macro `mensaje(linea)` de tal manera que el argumento `linea` se convierte en una cadena de caracteres y luego se muestra en pantalla.

g) Este problema ilustra el uso del operador «de concatenación» (##). Si la constante simbólica `COMPROBACION_ERROR` ha sido definida previamente, entonces se define la macro `mensaje(n)` de tal manera que se muestra en pantalla el valor de `mensajen` (por ejemplo, `mensaje3`).

14.37. *a*)
```
#if  defined(LOGICA)
    #define  VERDADERO  1
    #define  FALSO      0
    #undef   SI
    #undef   NO
#endif
```

o

```
#ifdef  LOGICA
    #define  VERDADERO  1
    #define  FALSO      0
    #undef   SI
    #undef   NO
#endif
```

b)
```
#if indicador == 0
    #define  COLOR  1
#elif indicador < 3
    #define  COLOR  2
#else
    #define  COLOR  3
#endif
```

c)
```
#if TAMANO == AMPLIO
    #define  ANCHO  132
#else
    #define  ANCHO   80
#endif
```

d)
```
#define  error(texto)  printf(#texto)
```

e)
```
#define  error(i)  printf("%s\n", error##i)
```

Índice

¡Estudia a tu propio ritmo y aprueba tu examen con Schaum!

Los Schaum son la herramienta esencial para la preparación de tus exámenes. Cada Schaum incluye:

- **Teoría de la asignatura con definiciones, principios y teoremas claves.**
- **Problemas resueltos y totalmente explicados, en grado creciente de dificultad.**
- **Problemas propuestos con sus respuestas.**



ISBN: 978-84-481-9846-6

www.mcgraw-hill.es

The McGraw·Hill Companies